高等学校计算机专业规划教材

基于Android平台的移动互联网应用开发（第2版）



雷擎　伊凡　编著

清华大学出版社

高等学校计算机专业规划教材

基于 Android 平台的移动互联网应用开发
（第 2 版）

雷　擎　伊　凡　编著

清华大学出版社
北　京

内 容 简 介

本书对Android应用程序开发的基本概念和技术进行了系统的讲解，并通过简单易懂的示例说明了其具体实现过程。通过本书的学习，读者可以牢固掌握Android编程技术的基本概念、原理和编程方法，通过实践的灵活运用，能够进行应用程序的实际开发。

全书分为三个部分，共10章。第一部分即第1章，详细介绍Android系统的体系结构、应用程序开发环境及调试环境的搭建；第二部分包括第2～4章，详细介绍用户界面的设计方法、常用布局、基本控件和高级控件、事件处理机制等实现Android用户界面的基本知识，以及用户浏览模式中菜单模式和动作条模式中各种应用的具体实现；第三部分包括第5～10章，详细介绍Android平台的高级知识，包括发送与接收消息、多任务与服务、实现应用程序的数据存储、访问数据资源的接口ContentProvider、触摸事件处理、定位服务和Google地图应用。

本书适合对Java编程有一定基础、希望掌握Android程序设计技术的读者，也适合作为高等学校计算机专业的教材，还可作为Android程序设计的培训教材。

图书在版编目(CIP)数据

基于Android平台的移动互联网应用开发/雷擎，伊凡编著. —2版. —北京：清华大学出版社，2017
(高等学校计算机专业规划教材)
ISBN 978-7-302-46976-6

Ⅰ. ①基…　Ⅱ. ①雷…　②伊…　Ⅲ. ①移动终端－应用程序－程序设计　Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2017)第078176号

责任编辑：龙启铭　张爱华
封面设计：何凤霞
责任校对：焦丽丽
责任印制：王静怡

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：23.75　　**字　　数**：551千字
版　　次：2014年6月第1版　　2017年8月第2版　　**印　　次**：2017年8月第1次印刷
印　　数：1～2000
定　　价：49.00元

产品编号：072325-01



关于本书

随着3G和触摸屏的技术发展,移动智能终端,即智能手机和平板电脑,已经成为人们日常通信和信息处理的工具,移动互联网正在改变人们的交流和生活方式。作为移动智能终端两大操作系统之一,Android的影响力已经渗透到移动领域以外,特别是物联网和电视等平台,Android应用程序也由个人应用逐步向企业应用扩展。掌握Android技术的人才就业前景非常广泛。

我们编写本书主要有两个目的:

全面系统地提供Android开发的基础知识,基于Android Studio开发环境提供编程示例。在本书的编写中把Android的基础知识,与自己的教学经验和学习的体会结合起来,希望能够引导Andorid技术学习者快速入门,系统地掌握Android编程技术。

由于目前Android技术更新很快,本书内容中的概念和原理主要参考Android的官方网站,尽量做到既准确又易于理解,代码示例均通过实际调试,可运行。

本书主要围绕Android技术,讲述如何利用Android相关技术,开发移动终端的互联网应用程序。全书共分为10章。

第1章概述使用Android技术在移动终端开发的基础知识,包括Android的基本常识和技术框架,并介绍如何搭建Android开发环境Android Studio和Android应用程序项目的结构。

第2章主要介绍Android的四大基础组件之一Activity、布局和资源的概念,介绍如何使用Android的Activity和布局管理器来设计移动终端的用户图形界面。

第3章主要介绍三部分内容。首先介绍Fragment的概念,Fragment的生命周期以及使用Fragment静态创建和动态创建用户界面的具体步骤和代码;第二部分主要介绍Android用户界面的事件处理机制,图形界面中的常用视图控件如何使用及如何进行事件处理,其中包括按钮控件中的Button、RadioButton、Checkbox和ToggleButton,Toast控件,以及文本控件中的TextView和EditText;最后使用例子说明了如何对界面进行处理,使其显示效果多样化。

第4章主要介绍了Android应用程序的浏览模式。从Android 3.0开始,Android系统的应用浏览模式发生了较大的改变,引入了向上和返回的设计原则,并且提供了相应的设计组件,其中包含菜单和动作条等。本章主要介绍菜单模式和动作条模式的实现。

第5章主要介绍Android系统的应用程序之间发送和接收消息的机制。介绍Android实现发送和接收消息的Intent、BroadcastReceiver组件和Notification组件的概念、用途和实现方法。

第6章主要介绍Android系统的多任务机制、主线程的概念和实现多任务的原理,以及在Androd系统中如何使用handler或AsyncTask实现应用程序的多任务。在多任务的基础上,本章的另一部分介绍Android的四大组件之一Service的概念和基本知识,以及在应用程序中实现Service的两种方式。

第7章主要介绍Android系统实现应用程序数据存储的机制,包括用户偏好的存取、文件的读取与保存和SQLite数据库的创建与操作。

第8章主要介绍Android的四大组件之一ContentProvider的概念和相关知识,以及如何创建和使用ContentProvider,如何通过数据绑定,使用适配器、视图对象和SQLite数据库的ContentProvider实现数据加载,最终向用户显示数据。

第9章主要介绍Android有关触摸屏的应用程序开发。包括触摸事件的定义、触摸事件的传递机制、触摸点移动的速率跟踪、多点触控、手势识别和拖放处理等。

第10章有两个部分。第一部分主要介绍Android应用程序如何通过GPS和Android网络位置提供器,获取位置信息,实现定位服务;第二部分主要介绍Android应用程序如何使用Google公司提供的Google Maps API,实现应用程序中的Goolge地图的功能。

本书基本囊括了Android技术体系中的基础部分,并使用短小易懂的例子详细说明了如何应用。本书的不足之处在于,由于时间和篇幅的原因,本书只是编写了Android技术中最基础的部分,Android技术中关于网络互联、动画、游戏、服务器和其他更深层次的应用等都没有涉及,不能全面覆盖Android技术,还请读者谅解。并且,由于水平的原因,在本书的编写过程中可能存在一些对Android技术及移动互联网技术介绍不全面或者表述疏漏的地方,敬请读者批评指正。

读者对象

本书是Android技术入门的基础类书籍,通过本书的学习读者可以牢固掌握Android编程技术的基本概念、原理和编程方法,能够进行应用程序的实际开发。

本书适合对Java编程有一定基础、希望掌握Android程序设计技术的读者,也适合作为高等学校计算机专业的教材,还可作为Android程序设计者的培训教材。

致谢

在本书的写作过程中，得到了很多人士的悉心帮助，在此谨向给予本书帮助的诸位及本书所参考的官方网站和网站社区表示诚挚的感谢。

特别感谢对外经济贸易大学信息学院和远程学院，为本书的教学和实践提供了支持平台。

特别感谢美国内华达大学拉斯维加斯分校（UNLV）的 Yingtao Jiang 教授和他的实验室，给予了本书很大的帮助和支持，并提出了中肯的意见。

特别感谢清华大学出版社的编辑们，他们为这本书的再版做了很多辛苦工作。

编　者

2017 年 5 月


目 录

第 9 章　触摸事件处理　/289

第 10 章　定位服务与 Google 地图　/327

附录 A　Eclipse 的 Android App 开发环境　/352

第 1 章

Android 开发基础

Android 是一种以 Linux 为基础的开放源代码操作系统，主要使用于便携设备。目前尚未有统一中文名称，中国大陆地区大多数人使用“安卓”或“安致”。Android 操作系统最初由 Andy Rubin 开发，主要用于支持手机。2005 年由 Google 收购注资，并组建开放手机联盟，进行开发改良，逐渐扩展到平板电脑及其他领域。2008 年，在 Google I/O 大会上，谷歌提出了 Android HAL 架构图，并在同年 9 月正式发布了 Android 1.0 系统，这也是 Android 系统最早的版本。到目前为止，最新的正式版本是 2016 年 8 月发布的 Android 7.0 Nougat(牛轧糖)。截止到 2016 年 4 月的前三个月数据统计，Android 系统的全球市场份额已经达到了 76%，在美国市场该期间内的份额为 67.6%，在中国大陆城市地区该期间内的市场份额为 78.8%。目前 Android 的主要竞争对手是 Apple 的 IOS。

1.1 Android 入门

Android 系统是基于 Linux 平台的智能手机操作系统。Android 是一个开源的平台，它由操作系统、中间件、用户界面和应用软件等部分组成。

Android 系统采用软件堆层(Software Stack，又名软件叠层)的系统架构，由多个程序组合来完成一个共同的任务。Android 的系统架构主要分为三个层次：底层以 Linux 内核为基础，由 C 语言开发，是移动设备的操作系统，只提供基本功能；中间层包括函数库(Library)和虚拟机(Virtual Machine)，由 C++ 开发；最上层是各种应用软件组成，包括通话程序、短信程序和游戏等。

Android 系统作为一个移动开放平台，提供了与 Apple 移动系统不同的生态环境。任何手机厂商都可以免费使用 Android 系统来定制自己的产品，而且 Android 系统提供了标准的 SDK，应用软件可以由第三方开发者独立完成，目前应用的开发语言使用 Java。

1.1.1 Android 简介

从 Andoird 1.5 版本开始，Android 选择使用甜点名称作为系统版本的代号，其版本的名称分别为纸杯蛋糕(Cupcake)、甜甜圈(Donut)、松饼(Eclair)、冻酸奶(Froyo)、姜饼(Gingerbread)、蜂巢(Honeycomb)、冰激凌三明治(Ice Cream Sandwich)、果冻豆(Jelly Bean)、巧克力(KitKat)、棒棒糖(Lollipop)、棉花糖(Marshmallow)、牛轧糖(Nougat)。而且版本名称的英文名首字母按照字母的顺序，从 C 开始到 N。目前最新的版本 Android 7.0 (API level 24) Nougat 的下一个版本，应该是以 O 作为首字母的甜点名称。

表 1.1 是 Android 各版本对应的名称和 API Level 所对应的市场份额，可以在程序开发和发布时作为参考。

表 1.1 Android 的各个版本

版　本	名　称	API Level	比例
2.2	Froyo	8	0.1%
2.3	Gingerbread	10	1.7%
4.0	Ice Cream Sandwich	15	1.6%
4.1	Jelly Bean	16	6.0%
4.2		17	8.3%
4.3		18	2.4%
4.4	KitKat	19	29.2%
5.0	Lollipop	21	14.1%
5.1		22	21.4%
6.0	Marshmallow	23	15.2%
7.0	Nougat	24	0

Google 以 7 天为一个周期对 Android 版本的使用情况进行统计，表 1.1 中的比例表示了 2016 年 8 月 1 日为止，Android 设备中的不同版本分布比例。根据表 1.1 的数据，图 1.1 显示了目前版本的使用分布。从图 1.1 可以看出，目前手机上用得最多的版本为 4.4 的 KitKat 和 5.0、5.1 的 Lollipop，但 Marshmallow 已经占有了不少的份额，如果要发布应用，需要选择适合的 Android 版本，方便在不同的版本中支持该应用。

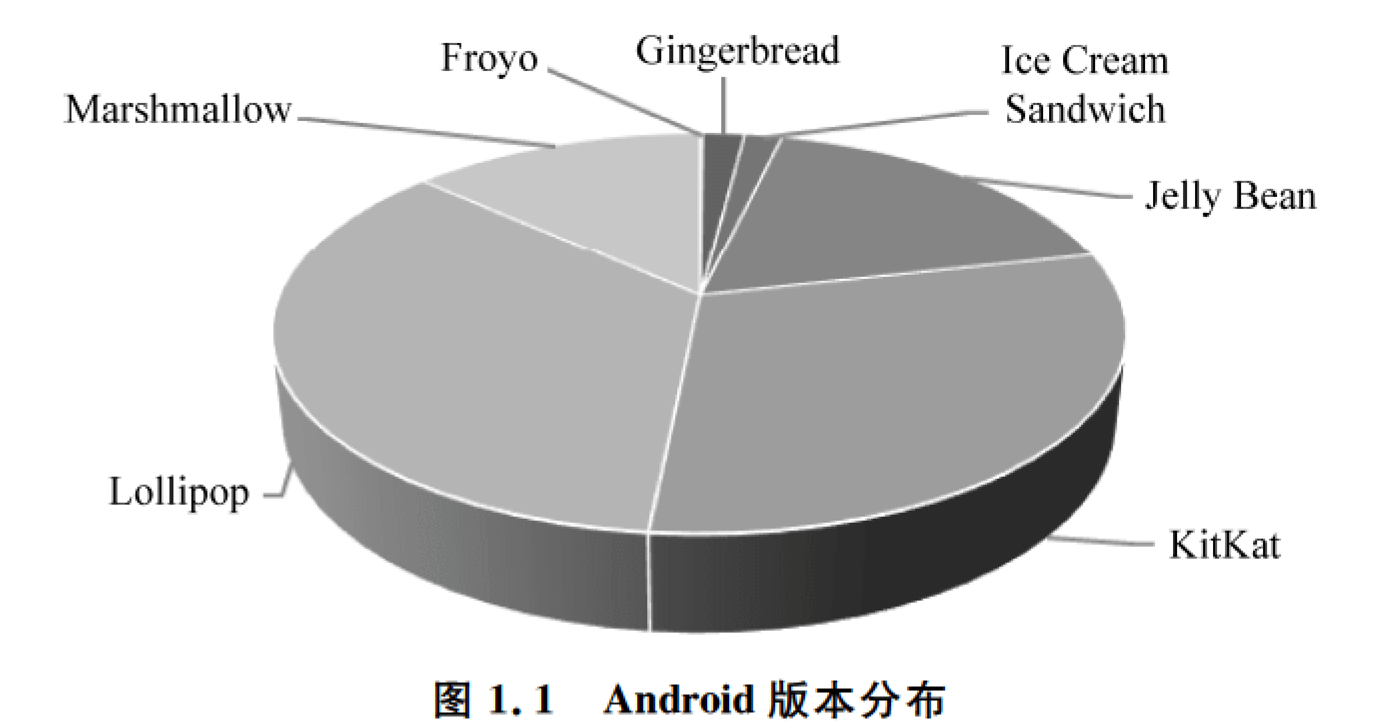
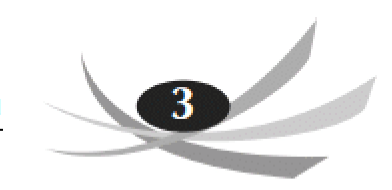


图 1.1 Android 版本分布

1.1.2 Android 技术架构

Android 技术架构见图 1.2。Android 技术架构由五部分组成，从顶层到底层，依次是 Application、Android Framework、Native Libraries、Android Runtime（ART）、



Hardware Abstraction Layer(HAL)、Linux Kernel。如果是Android应用开发者，只需要了解Application和Android Framework上面两个层次；如果是嵌入式和硬件移植的开发者，还需要了解Native Libraries、Android Runtime(ART)、Hardware Abstraction Layer (HAL)和Linux Kernel这几个部分。

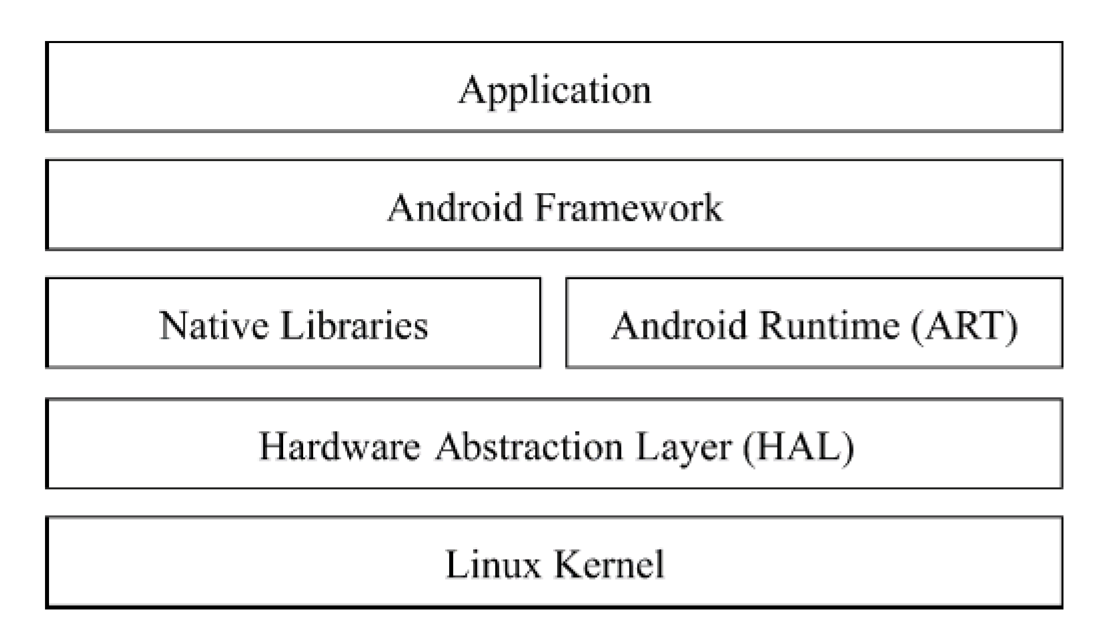


图1.2　Android技术架构

1. Application

Android配置一个核心应用程序集合，包括电子邮件客户端、SMS程序、日历、地图、浏览器、联系人、照相机、E-mail、时钟、拨号和其他设置。所有应用程序都是用Java编程语言写的。用户自己开发的App应用也在这个层次。

2. Android Framework

Android提供开放的应用框架Android Framework，应用开发者可以方便地设计丰富和新颖的应用程序。通过应用框架，开发者能够自由地利用设备硬件、使用访问位置信息、运行后台服务、设置闹钟，并且向状态栏添加通知等功能。由于这个应用开发框架是完全开放的，应用开发者能够使用框架的核心功能。

设计应用框架的目的在于方便组件的重用，简化应用程序的开发，而且应用程序都可以发布和使用应用框架中的功能。应用框架主要是由下面的功能组成的：

- 视图系统(View System)——包括丰富的、可扩展的视图集合，可用于构建一个应用程序，包括列表、网格、文本框、按钮，甚至是内嵌的网页浏览器。
- 内容提供者(ContentProviders)——使应用程序能访问其他应用程序(如通讯录)的数据，或共享自己的数据。
- 管理器(Manager)——包括许多管理器。其中资源管理器(Resource Manager)为应用程序提供可访问的非代码资源，如本地化字符串、图形和布局文件等；通知管理器(Notification Manager)支持所有的应用程序能在状态栏显示自定义警告；活动管理器(Activity Manager)管理用户界面应用程序的生命周期，提供通用的导航回退功能；位置管理器(Location Manager)支持移动设备上基于位置和地图的应用，可以和Google Location Services API配合使用，实现更强大的框架。

3. Native Libraries

Native Libraries(本地库)是由Android提供的一系列本地头文件和共享库文件，应用程序通过Android Framework调用。Native Libraries支持使用硬件传感器、访问存储

器、处理用户输入、配置信息设置等功能。随着 Android 版本和 Android API Level 的不断更新，Native Libraries 的功能逐渐增长。到目前为止，包括 C/C++ 库、ZLib 压缩库、动态链接库、嵌入式 3D 图形加速标准 OpenGL ES 库、分配和管理 OpenGL ES 的 EGL 库、嵌入式音频加速标准 OpenSL ES 库、FreeType 位图和矢量字体处理库、强大而轻量级的关系数据库引擎 SQLite、WebKit 和 OpenMAX AL 等库。

4. Android Runtime

Android Runtime(ART)包括核心库、ART 和 Dalvik 虚拟机。

从 Android 4.4 开始，Adnroid 就推出了新的 Android 运行时(ART)，替代之前版本的 Dalvik。ART 的功能是管理和运行 Android 应用程序和 Android 的一部分系统服务。与 Dalvik 相比，ART 提供了许多新的功能，来改善 Android 平台和应用的性能。ART 和它的前身 Dalvik 都是专门为 Android 项目创建的，ART 遵循 Dalvik 可执行格式和 Dex 字节码规范。

ART 和 Dalvik 运行 Dex 字节码时是兼容的，因此基于 Dalvik 开发的应用也可以在 ART 上运行。但有时候，基于 Dalvik 的技术无法在 ART 上实现。

5. Hardware Abstraction Layer

硬件抽象层(Hardware Abstraction Layer，HAL)为硬件厂商定义了一个标准接口，Android 系统通过这个标准接口使用硬件实现的功能，底层的硬件驱动实现对于 Android 系统透明。也就是说，Android 的上层应用只需调用这些标准接口来实现软件的功能，不必了解具体是哪家厂商的产品，如何实现。

HAL 使得 Android 系统的上层软件功能与硬件实现隔离开，不同硬件的支持，不会影响也不需要修改上层软件系统。

当然，针对不同的厂商产品，需要开发对应的 HAL。HAL 的实现部分以.so 的文件模式存储在共享库模块中。

6. Linux Kernel

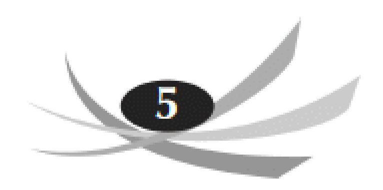
Android 的 Linux Kernel 提供操作系统的核心系统服务，包括安全管理、内存管理、进程管理、网络堆栈、电源管理和驱动模型等。例如声音、显示、相机、蓝牙、Wi-Fi 等设备的驱动，都在这一层实现。

1.2 Android Studio 环境搭建

在进行 Android 应用程序开发之前，需要搭建 Android 应用程序开发环境。本书中采用开源的 Android Studio 2.1.2 集成开发环境开发工具。有关 Eclipse 平台开发 Android 的开发环境搭建，参看附录 A。

Android Studio 是 Android 开发的官方集成开发环境(IDE)，提供开发 Android App 所需要的各种支持工具和软件包。安装完的 Android Studio，具有下面的功能：

- IntelliJ IDE+Android Studio plugin；
- Android SDK 工具包；
- Android 平台工具；



- Android 开发平台；
- 包含 Google Play Service、带有 Android 系统图像的 Android 模拟器。

由于 Android Studio 把 Android 应用程序开发所需的运行环境、库、开发工具和界面都集成为一个包，因此安装比较简单。具体安装过程见 1.2.1 节。如果已经熟悉 Android Studio 开发环境的安装，可以跳过 1.2 节，直接阅读后面的内容。

Android Studio 针对 Windows、Mac、Linux 不同的操作系统，提供相应的安装包。具体的下载网址为 https://developer.android.com/studio/index.html。

在下载完成 Android Studio 安装包之后，首先需要确认操作系统已经安装了 JDK，并且版本在 1.8 以上。下面针对不同操作系统，介绍 Android Studio 具体的安装步骤。

1.2.1　基于 Windows 的安装

具体的安装步骤如下：

(1) 打开命令行窗口，输入 javac -version 命令，如果 JDK 不存在，或 JDK 的版本低于 1.8，从网址 http://www.oracle.com/technetwork/java/javase/downloads/ jdk8-downloads-2133151.html 下载 Java SE Development Kit 8，并进行安装。

(2) 运行所下载的 Android Studio 安装包.exe 文件。

(3) 根据安装提示安装 Android Studio 和 SDK 工具。

(4) 在安装完成界面，选中 Start Android Studio 选项，单击 Finish 按钮，启动 Android Studio 设置，进行初始设置。

(5) 根据需要选择 Android Studio 预定的设置，或者选择自定制的设置，然后单击 OK 按钮，见图 1.3。建议选择默认选项，即位于下面的 Android Studio 预定设置选项，继续下面的配置。这个过程包括下载 Android 的组件和工具包，并进行安装，需要的时间会有点长，请耐心等待，直到完成配置过程，出现 Android Studio 启动界面，见图 1.4。关闭 Android Studio 启动界面，就完成了安装。

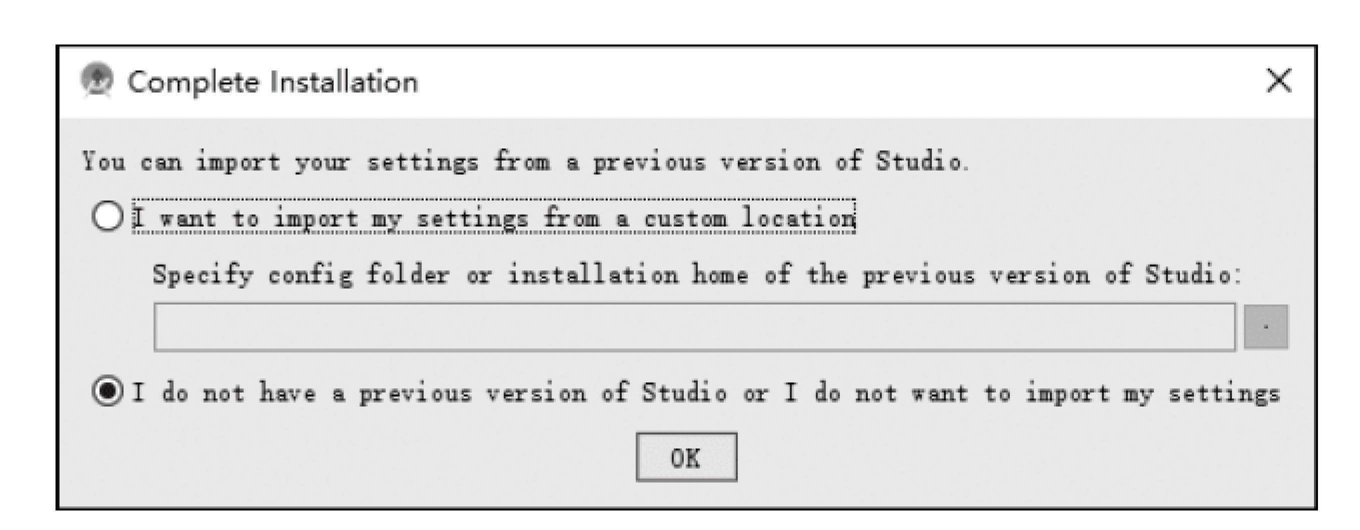


图 1.3　配置选择

(6) 设置 Windows 系统环境变量。注意，在某些 Windows 系统中，Android Studio 安装脚本找不到 JDK 的路径，如果遇到这种情况，需要在环境变量中设置 JDK 路径。具体操作如下：选择 Start→Computer→System Properties→Advanced System Properties，然后选择 Advanced→Environment Variables，增加一个新的系统变量 JAVA_HOME 指定 JDK 路径，例如，C:\Program Files\Java\jdk1.8.0_77。

到此，基于 Windows 的 Android Studio 安装完成，下一步可以进行创建 Android 项



图 1.4　Android Studio 启动界面

目的工作了。

1.2.2　基于 Mac 的安装

具体的安装步骤如下：

(1) 打开命令行窗口，输入 javac -version 命令，如果 JDK 不存在，或 JDK 的版本低于 1.8，从网址 http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html 下载 Java SE Development Kit 8，并进行安装，见图 1.5。

(2) 运行所下载的 Android Studio DMG 文件。

(3) 把 Android Studio 拖曳到应用 Application 文件夹，然后运行 Android Studio。

(4) 自己根据需要选择 Android Studio 预定的设置，或者选择自定制的设置，然后单击 OK 按钮。

(5) 根据 Android Studio 的设置提示，完成包括下载 Android SDK 组件等在内的其余的设置。



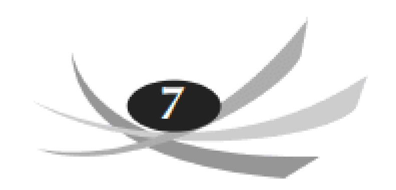
图 1.5　安装基于 Mac 的 Android Studio

安装过程中的一些细节，可以参考 Windows 安装的过程。到此，基于 Mac 的 Android Studio 安装完成，下一步可以进行创建 Android 项目的工作了。

1.2.3　基于 Linux 的安装

具体的安装步骤如下：

(1) 打开命令行窗口，输入 javac -version 命令，如果 JDK 不存在，或 JDK 的版本低



于 1.8，从网址 http://www.oracle.com/technetwork/java/javase/downloads/ jdk8-downloads-2133151.html 下载 Java SE Development Kit 8，并进行安装。

(2) 在合适的目录下，例如/usr/local 目录，将所下载的 Android Studio 安装包.zip 文件解压缩。

(3) 运行 Android Studio，打开 Terminal，进入 android-studio/bin/目录，执行 studio.sh，见图 1.6。建议把 android-studio/bin/的路径加入 PATH 环境变量，以便在任何目录下都可以启动 Android Studio。

```
Terminal
File Edit View Search Terminal Help
smain@smain:~$ sudo mv ~/Documents/Downloads/android-studio /usr/local/
smain@smain:~$ cd /usr/local/android-studio/bin
smain@smain:/usr/local/android-studio/bin$ ./studio.sh
log4j:WARN No appenders could be found for logger (io.netty.util.internal.l
ogging.InternalLoggerFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for mo
re info.
```

图 1.6 Linux Terminal 中的命令

(4) 自己根据需要选择 Android Studio 预定的设置，或者选择自定制的设置，然后单击 OK 按钮。

(5) 根据 Android Studio 的设置提示，完成包括下载 Android SDK 组件等在内的其余的设置。

(6) 安装过程中的一些细节，可以参考 Windows 安装的过程。到此，基于 Linux 的 Android Studio 安装完成，下一步可以进行创建 Android 项目的工作了。

1.3 第一个 Android 应用程序

1.3.1 创建 Android 项目

在完成了 Android 开发环境的安装和配置后，就可以使用 Android Studio 开始开发 Android 应用程序了。下面介绍第一个 Android 应用程序——Hello Mobile World——的开发和运行过程，它的功能是在界面上显示“Hello World”的字符。

第一次编写 Android 应用程序需要以下四个步骤：

- 创建一个新的 Android 项目。
- 运行应用程序。
- 定义简单的用户界面。
- 启动另一个 Activity。

(1) 在 Windows 下运行 Android Studio，出现图 1.7 所示的 Android Studio 启动界面后，选择第一项 Start a new Android Studio project。

(2) 在新项目的配置界面，填写 Application name、Company Domain 和 Project location 对应的文本框，见图 1.8。

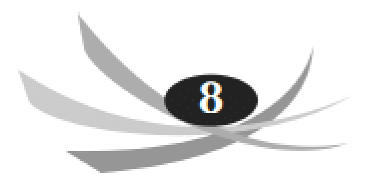




图 1.7　Android Studio 启动界面

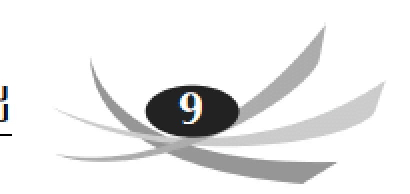
图 1.8　项目配置界面

其中，应用名称 Application name 指计划要开发的 App 名称，这里的 App 名称为“HelloWorld”；公司域名 Company Domain 会体现在 App 的 Java 包的配置上，Package 的名称刚好与其相反，并反向对应 Java 源代码存储的目录；项目位置 Project location 是指整个项目所在的路径，可以通过文本框后面的按钮，在计算机中选择合适的存储位置。

（3）在选择 Android 运行的设备的界面上，选择 Phone and Tablet 复选框，让 App 的运行环境设置为手机和平板电脑，见图 1.9。

如果在后面的开发中，要开发可穿戴设备、电视、音频和眼镜方面的应用，可以根据需要选择对应的复选框，来运行和调试 App。

（4）在 Add an Activity to Mobile 界面，选择默认选项 Empty Activity，单击 Next 按钮。图形用户界面是通过类 Activity 的子类来实现的，这些子类统称为 Activities。



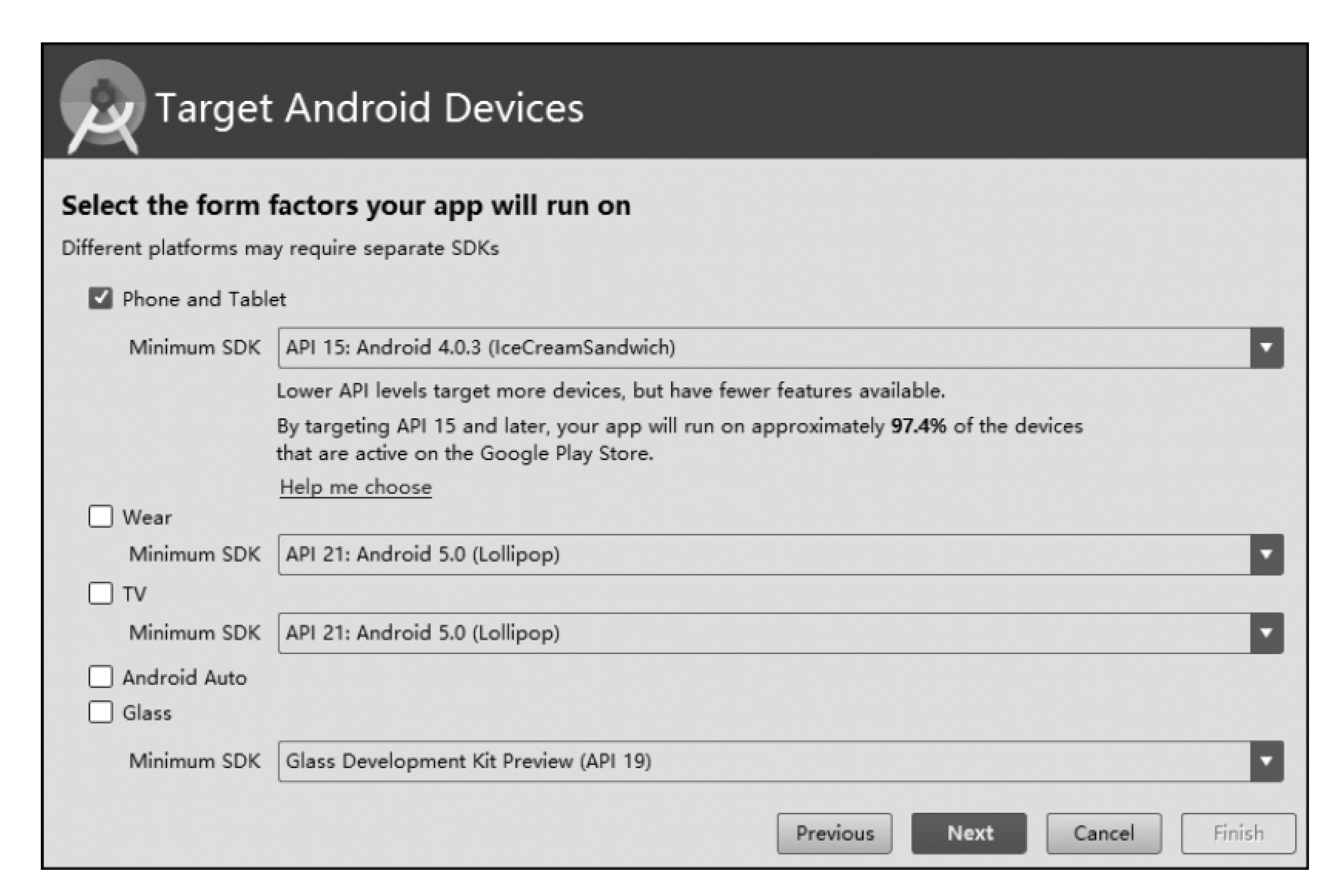


图 1.9　选择 Android 运行的设备

(5) 在 Activity 配置界面，填写 Activity Name、Layout Name 对应的文本框，见图 1.10。

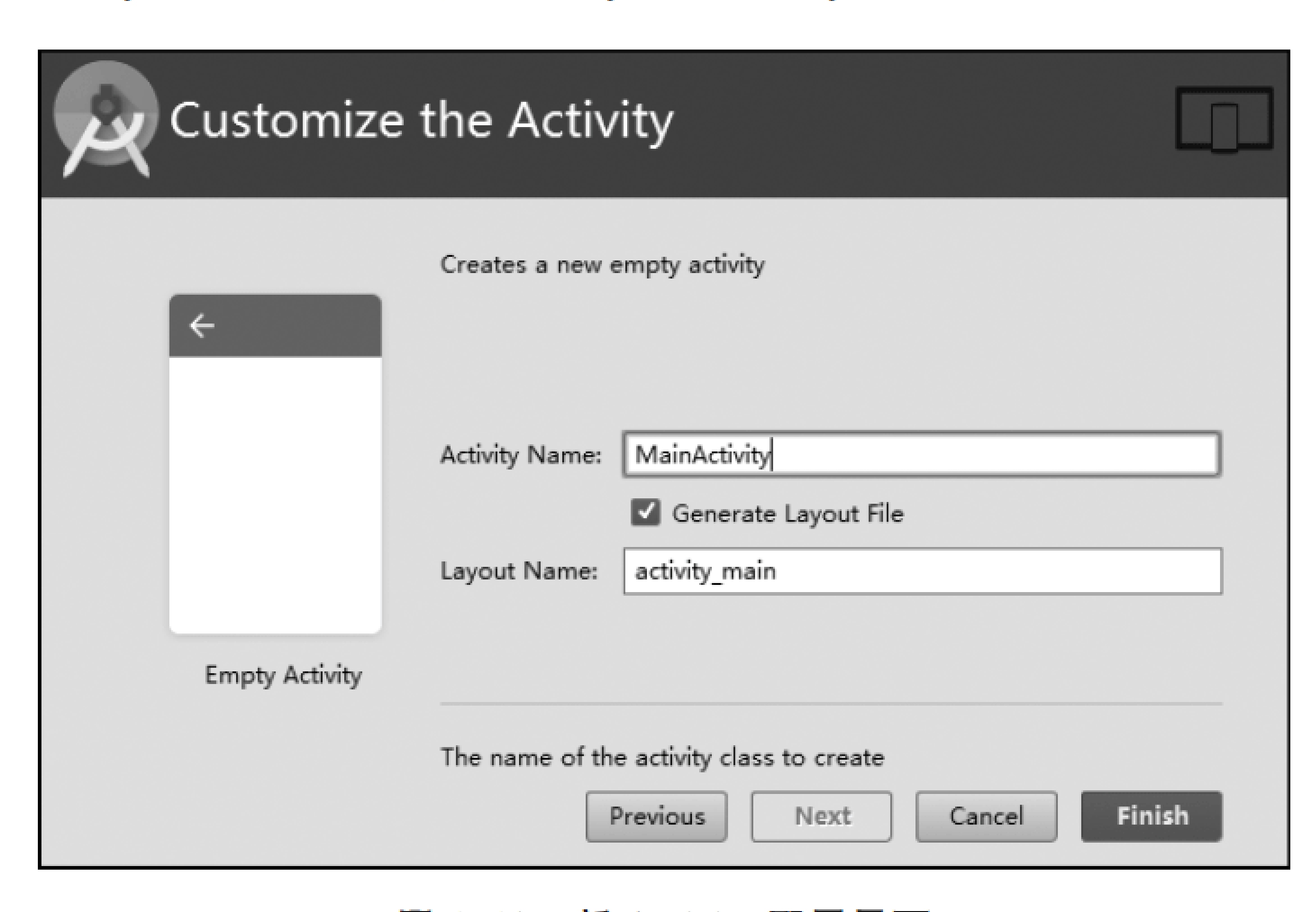


图 1.10　新 Activity 配置界面

其中，Activity Name 对应 Java 源代码中 Activity 子类的名称；Layout Name 指 XML 布局文件的名称，也就是说，Android 系统可以通过 XML 文件，设置 Android 手机和平板电脑的用户界面具体有哪些组件，如何显示。在这里，选用 Android Studio 默认的名称。

(6) 单击 Finish 按钮，完成 Android Studio 新项目创建的设置，就可以看到 Android Studio 开发界面了。

(7) 关闭提示对话框，单击开发界面左边界的 Project 标签，打开 Project 的内容，在左边的窗口可以看到刚才所创建项目的目录结构。展开相应的目录，双击 java 目录下的

MainActivity，以及 res/layout 目录下的 activity_main. xml 文件，在右边的窗口可以看到文件的内容，见图 1.11。

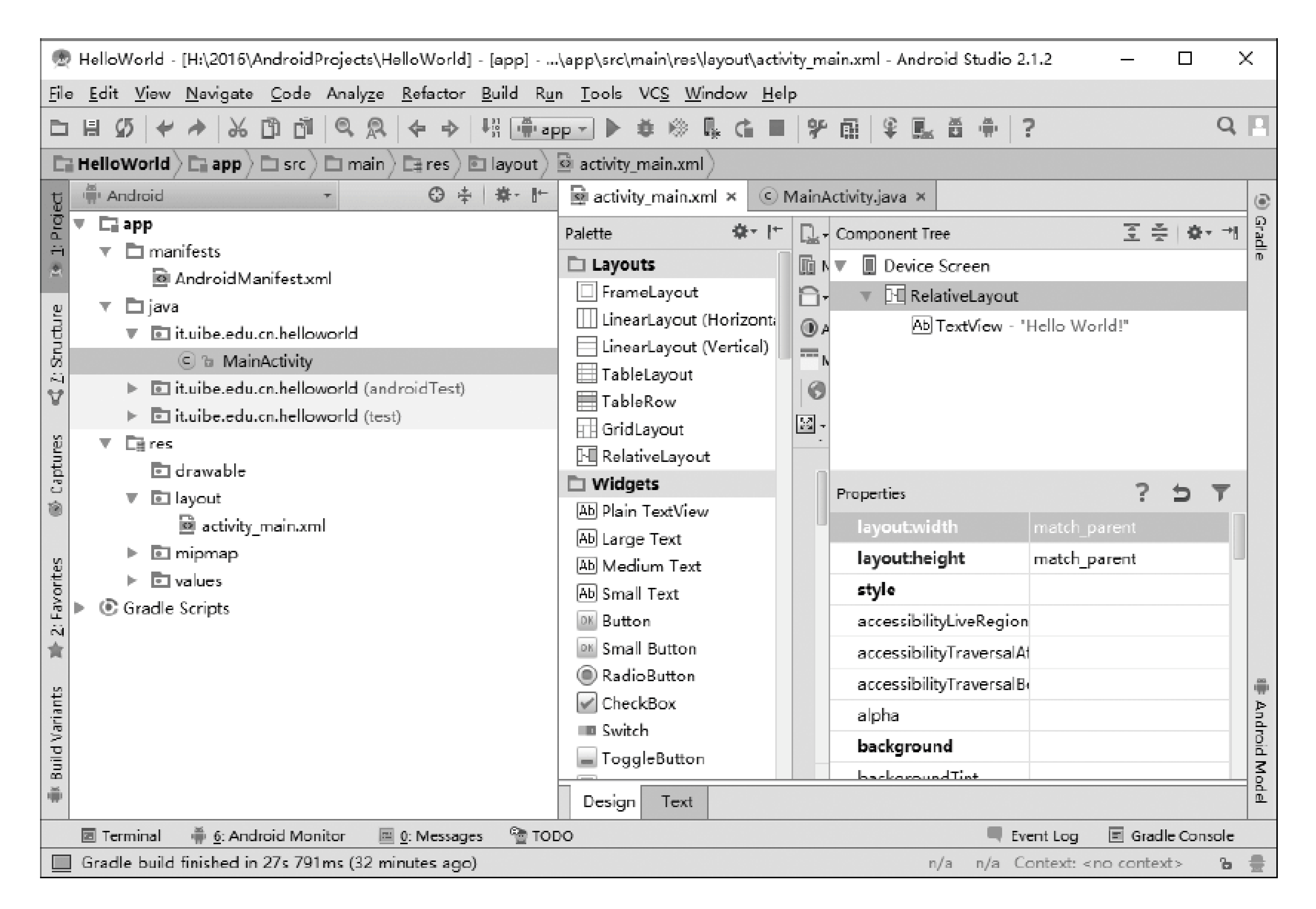


图 1.11　Android Studio 开发界面

到这里，就完成了一个新项目的创建。在这个新创建的 HelloWorld App 项目中，默认创建了一些缺省文件，分别存放在不同的目录下，见图 1.11。下面对其中两个重要的文件 MainActivity. java 和 activity_main. xml 进行一下说明。

代码 1.1 是 java 目录下的 MainActivity. java 文件的具体内容。

代码 1.1　MainActivity. java 文件

```
package it.uibe.edu.cn.helloworld;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

从代码 1.1 中可以看出 MainActivity 是 Activity 的子类。Activity 是 Android 系统中用于实现用户图形界面的类。

protected void onCreate()方法是 Activity 所定义的方法，当 Activity 启动时由 Android 系统调用，首先在这个方法中执行初始化和用户界面设置工作，Java 源程序的 setContentView()方法，通过 R. layout. activity_main 类调用了 activity_main. xml 定义的用户界面，运行时在屏幕上显示“Hello World!”。一个 Activity 并不一定要有用户界面，但通常都会有。Activity 概念可以参照 Applet 来理解。

代码 1.2 是 activity_main. xml 文件的具体内容。

代码 1.2　activity_main. xml 文件

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="it.uibe.edu.cn.helloworld.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</RelativeLayout>
```

在这个 XML 布局文件中，元素 TextView 下的 android:text="Hello World!"语句定义了屏幕显示的内容为“Hello World!”。该布局文件的其他功能和具体代码的含义在后续章节解释。

进行完上一步的工作后，就可以运行 App 了。

Android App 的运行可以在两个不同的环境中进行选择：真正的 Android 设备和 Android 模拟器。接下来详述 Android 设备和 Android 模拟器上安装和运行 Android App 的具体过程。

1.3.2　在手机上运行 HelloWorld App

下面以 Samsung SM-A8000 型号的手机为例，进行环境配置，运行 App，其他的手机配置过程类似。

（1）通过 USB 将手机连接到计算机。

将用于测试的手机用 USB 连线连接到计算机。

（2）设置手机的 USB 调试选项。

打开手机的设置界面，打开“开发者选项”，见图 1.12 左图；然后在打开后的“开发者选项”配置界面中，将第一项设置为“开”的状态，“USB 调试”设置为“开”的状态，见图 1.12



图 1.12　手机的 USB 调试选项设置

右图。

（3）运行 Android Studio 中的应用程序 HelloWorld App。

在 Android Studio 中，单击图 1.13 中所示的 App 运行按钮 ▶，尝试运行 HelloWorld App 程序。因为这是第一次运行 App，所以平台会给出运行设备选择的对话框，请先进行运行设备的选择或配置。

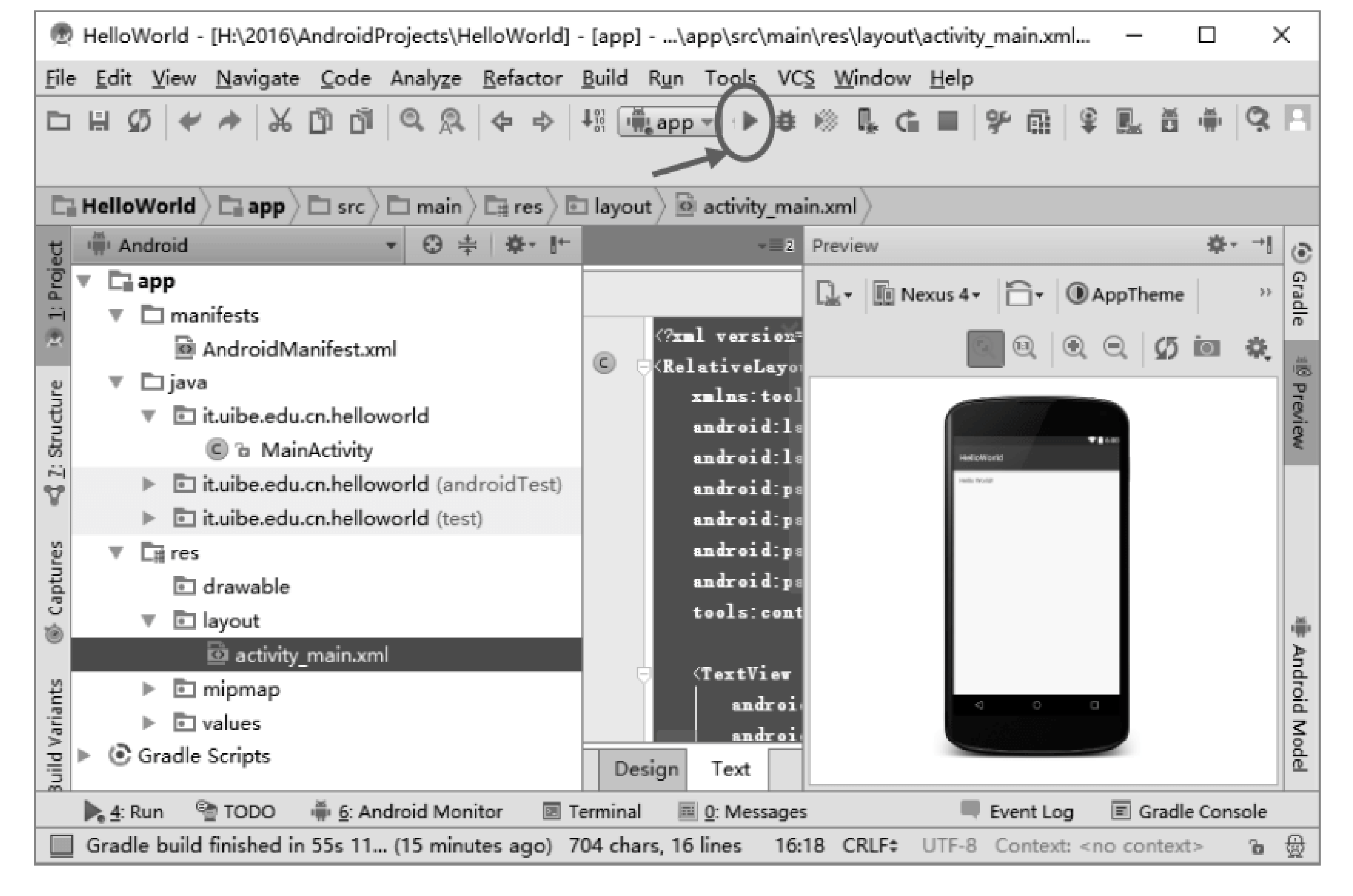


图 1.13　Android Studio 开发界面

（4）选择运行设备。

因为前面已经设置了手机的开发者 USB 调试选项，成功与设备连接后，在运行设备选择对话框中会出现所连接的手机选项，见图 1.14。

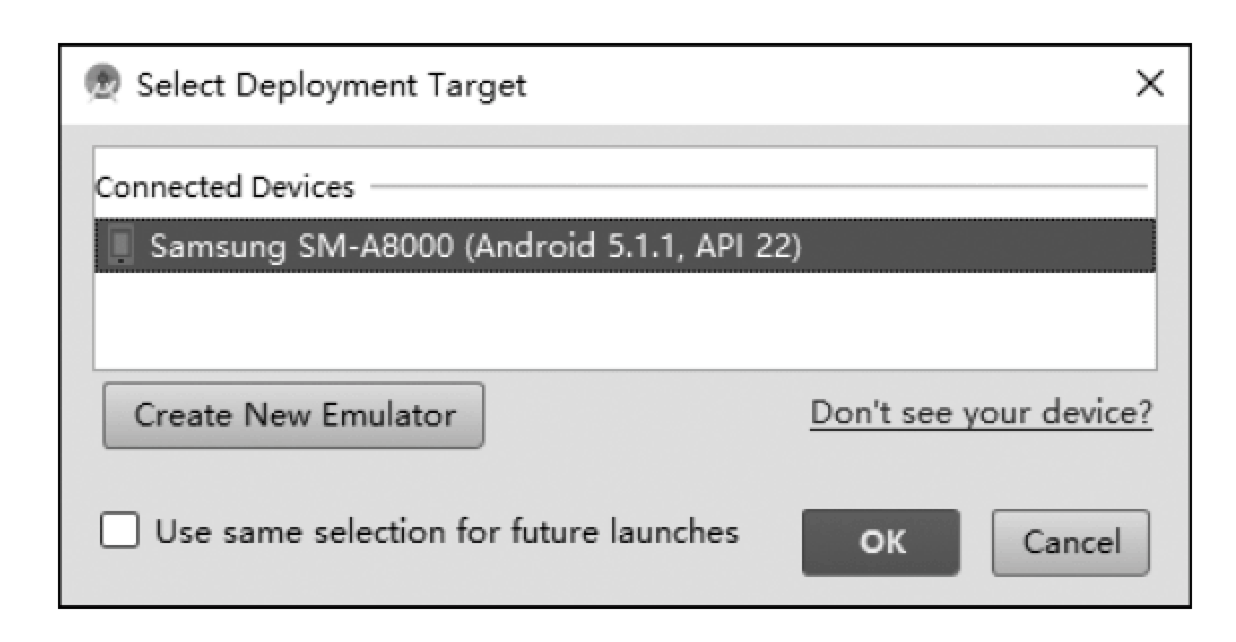


图 1.14　选择设备对话框

选择手机设备作为运行设备，单击 OK 按钮，在手机上运行 HelloWorld App。

(5) App 在手机上安装运行。

在手机上运行 App 的速度很快，几乎马上就可以从手机上看到图 1.15 左图所示的结果。从 App 返回手机的主界面，可以看到屏幕上出现一个绿色小机器人，表明 HelloWorld App 已经装载到了手机上，可以单击其图标。



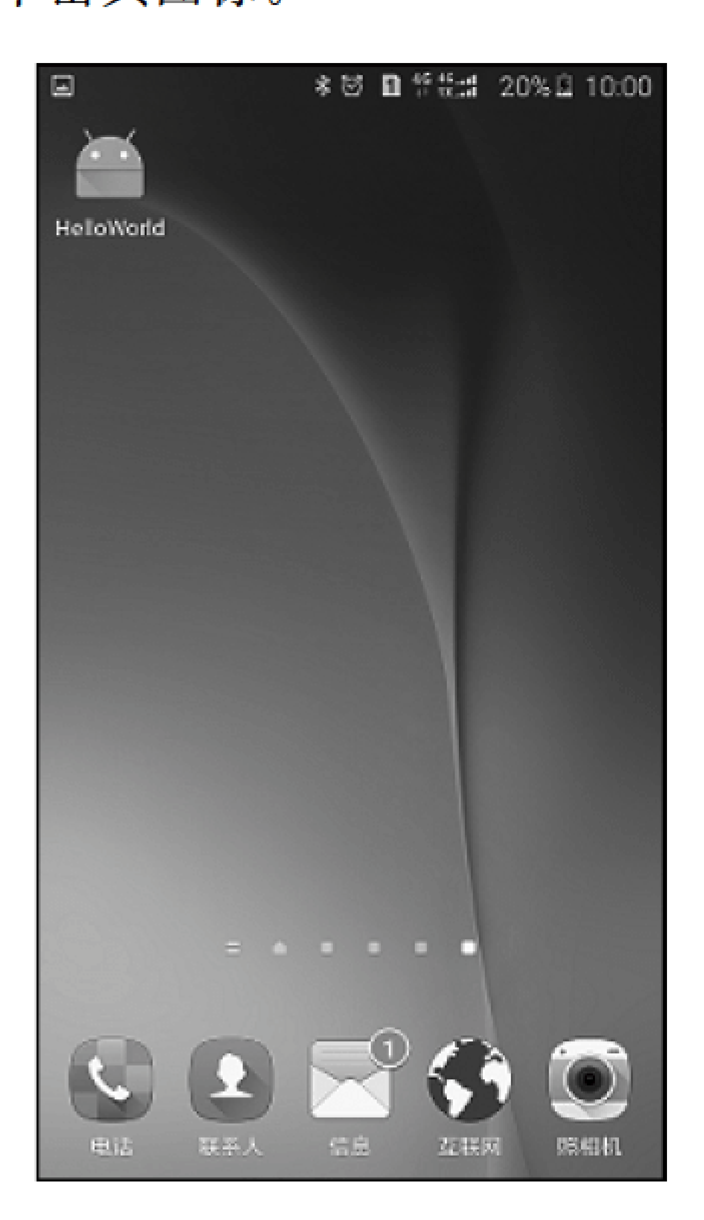


图 1.15　手机运行结果

在手机上执行 App，实际上是把 Android Studio 编译后的 apk 文件装载到手机并执行的过程，相当于从网上下载应用，并安装后执行的过程。在开发过程中，可以一直使用手机来进行开发 App 的调试和测试，但大多数情况下，会使用 Android 手机模拟器，即 AVD 来进行调试、运行和测试。因为 AVD 可以根据需要设置成不同 Android 版本、不同显示模式、不同性能、不同厂商的模拟运行环境，对 App 进行多方面的测试，有利于所开发 App 的兼容性、健壮性和适应性。

1.3.3 在 AVD 上运行 HelloWorld App

不对上面的程序做任何修改，下面在模拟器上运行该 App。Android 模拟器能够模拟 App 在真实手机上的运行效果和功能，在开发过程中支持程序员对 Android App 进行代码测试、调试和运行。

如果刚才在手机上运行了 HelloWorld App，首先要断开与手机的 USB 连接，然后再进行下面的步骤。

(1) 在创建 AVD 之前，需要先安装 HAXM，即 Hardware Accelerated Execution Manager，Intel 的硬件加速执行管理器。

选择 Tool→Android→SDK Manager，单击对话框中的蓝色链接 Launch Standalone SDK Manager，打开独立的 Android SDK Manager，将右边的滚动条拖曳到最底部，可以看到最后一个选项 Intel x86 Emulator Accelerator (HAXM installer)，见图 1.16。

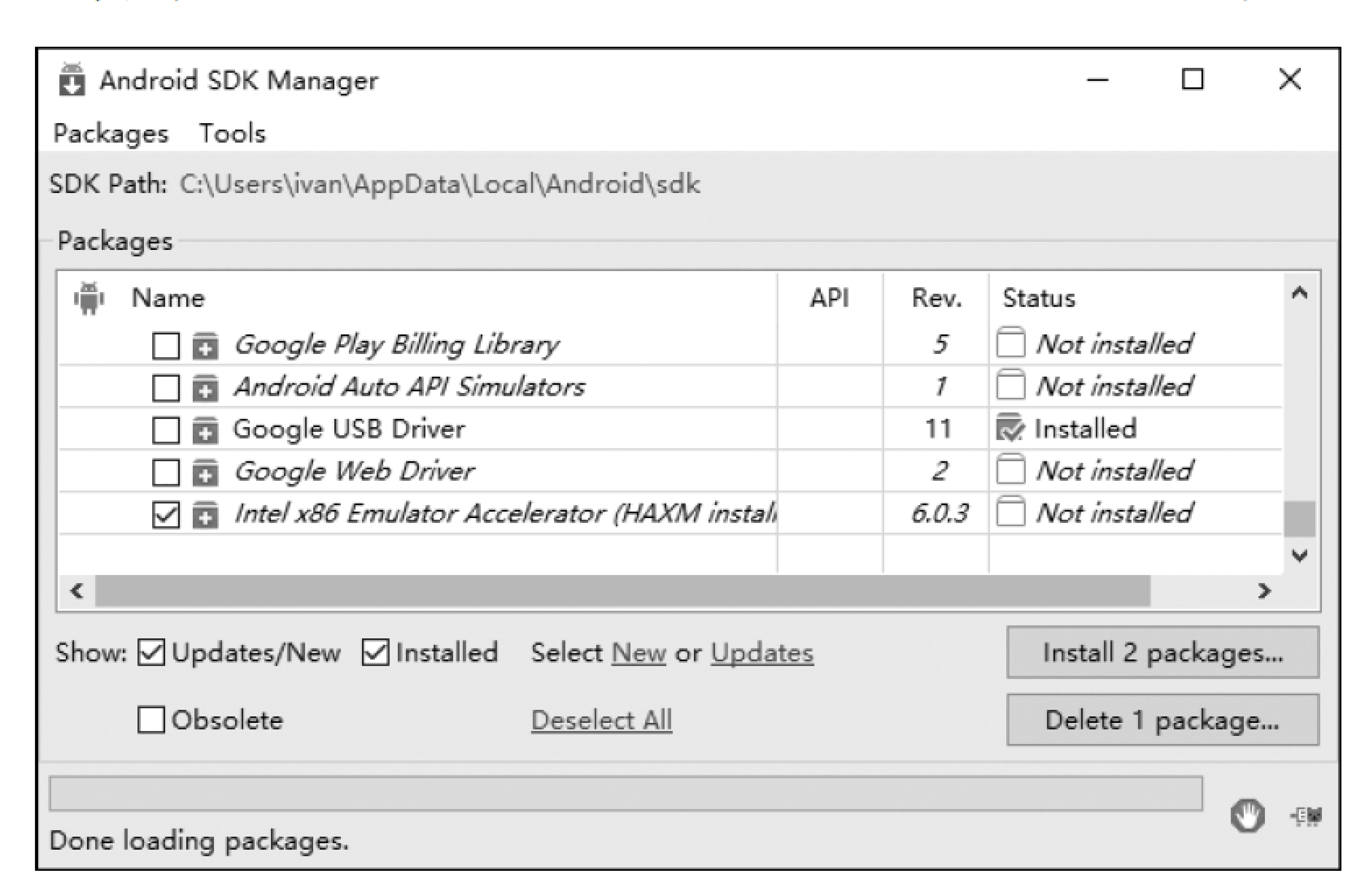


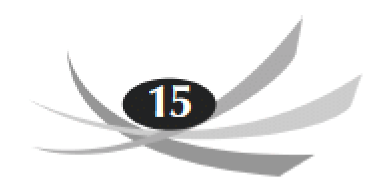
图 1.16 Android SDK Manager

选中该选项，然后向上拖曳滚动条，同时选中所需要 API level(例如 API 23)中的 Intel x86 Atom System Image 或 Intel x86 Atom_86 System Image 选项，单击右下的按钮 Install 2 packages，安装这两个包。

要注意的是，虽然安装完成，在 Status 栏显示 Installed，但 HAXM 并没有真正安装到系统，还需要进入 HAXM 安装包所在的目录手动安装 HAXM。具体路径是 Windows 用户目录下的 AppData\Local\Android\sdk\extras\intel\Hardware_Accelerated_Execution_Manager。

(2) 单击 Android Studio 的运行按钮，运行 Android Studio 中的应用程序 HelloWorld App。

(3) 在图 1.14 所示的选择设备对话框中单击 Create New Emulator 按钮，创建运行 App 的模拟设备 AVD(Android Virtual Device)。



(4) 在接下来的对话框中选择模拟的手机型号、Android 的版本及 API level，进行模拟器配置，然后在最后一个对话框中检查各项配置是否正确，在 AVD Name 对应的文本框中给出 AVD 的名称，单击 Finish 按钮，完成 AVD 创建，见图 1.17。第一次配置可以采用默认设置。

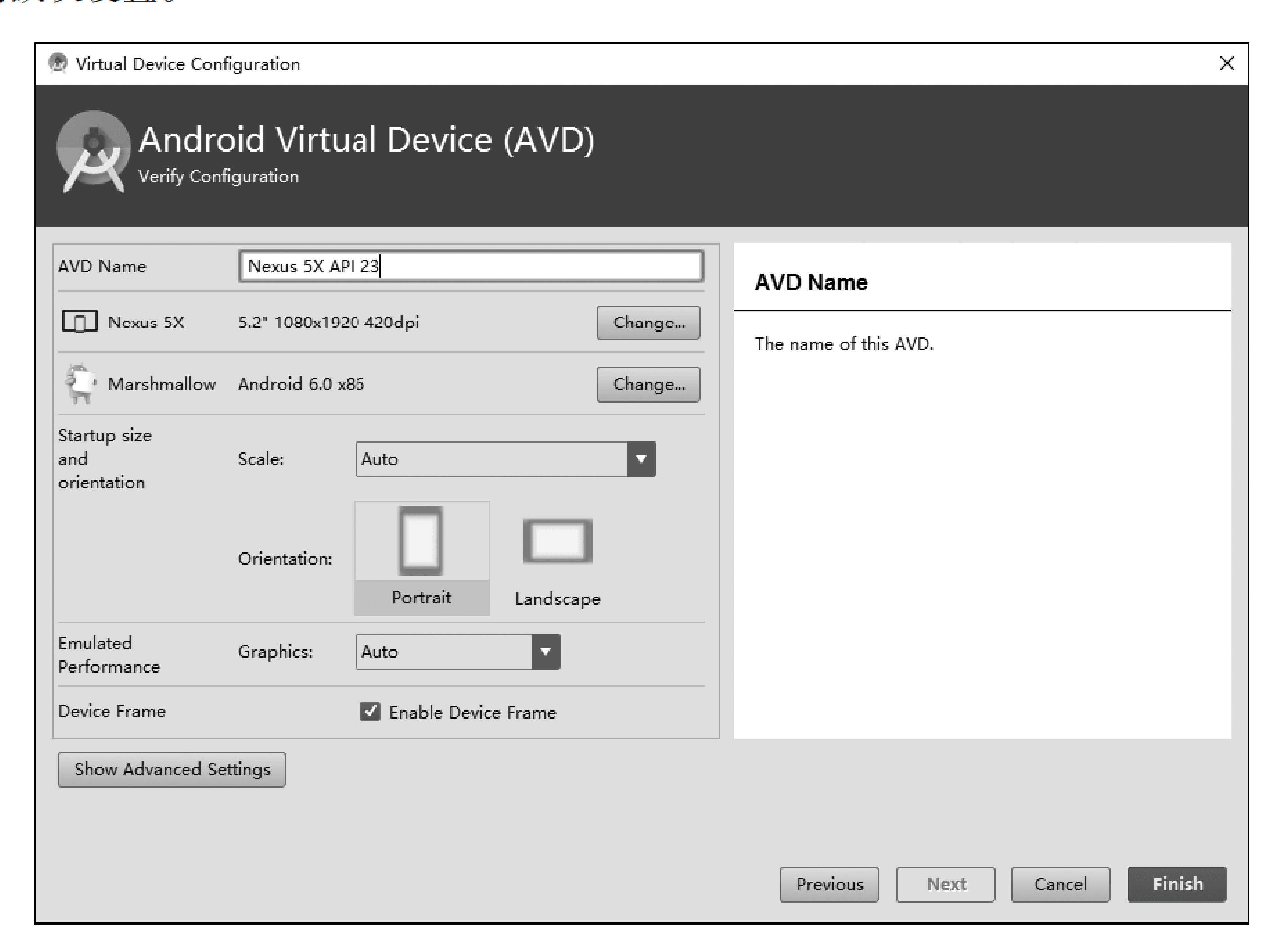


图 1.17　AVD 配置

(5) 在图 1.14 所示的选择设备对话框中，双击刚才所创建的 AVD，Android Studio 会把 HelloWorld App，也就是 HelloWorld.apk 文件，安装到 AVD 上并运行。

这时需要真正耐心地等待运行结果。一般来说，第一次在仿真器上运行 Android 应用程序需要花很长时间。具体的时间与机器性能和配置有关。

如果运行 3～4 分钟却只看到仿真器上的 Android 在闪烁，见图 1.18 左图，此时不要着急，请耐心等待。

几分钟之后，可以看到图 1.18 中图所示的运行结果显示在屏幕上。图 1.18 右图的工具条是对模拟器手机进行操作的各种功能按钮，可以实现关机、调节声音大小、摇动、照相等功能。如果单击 O，就回到手机 Home 的状态，通过单击应用程序图标，可以看到手机上装载的应用程序，从中找到刚才所运行的 HelloWorld App 的白底绿色小机器人图标。

(6) AVD 可以创建多个，如果要在后续的 App 开发过程中默认使用某个 AVD，在双击它之前，选中图 1.14 中的 Use same selection for future launches 复选框。

Android Studio 的 AVD 除了可以在运行 App 时在选择设备配置时创建，还可以在其提供的 AVD 管理工具里进行创建、删除、修改等操作，具体的操作通过选择 Tool→Android→AVD Manager，在弹出的对话框中进行。

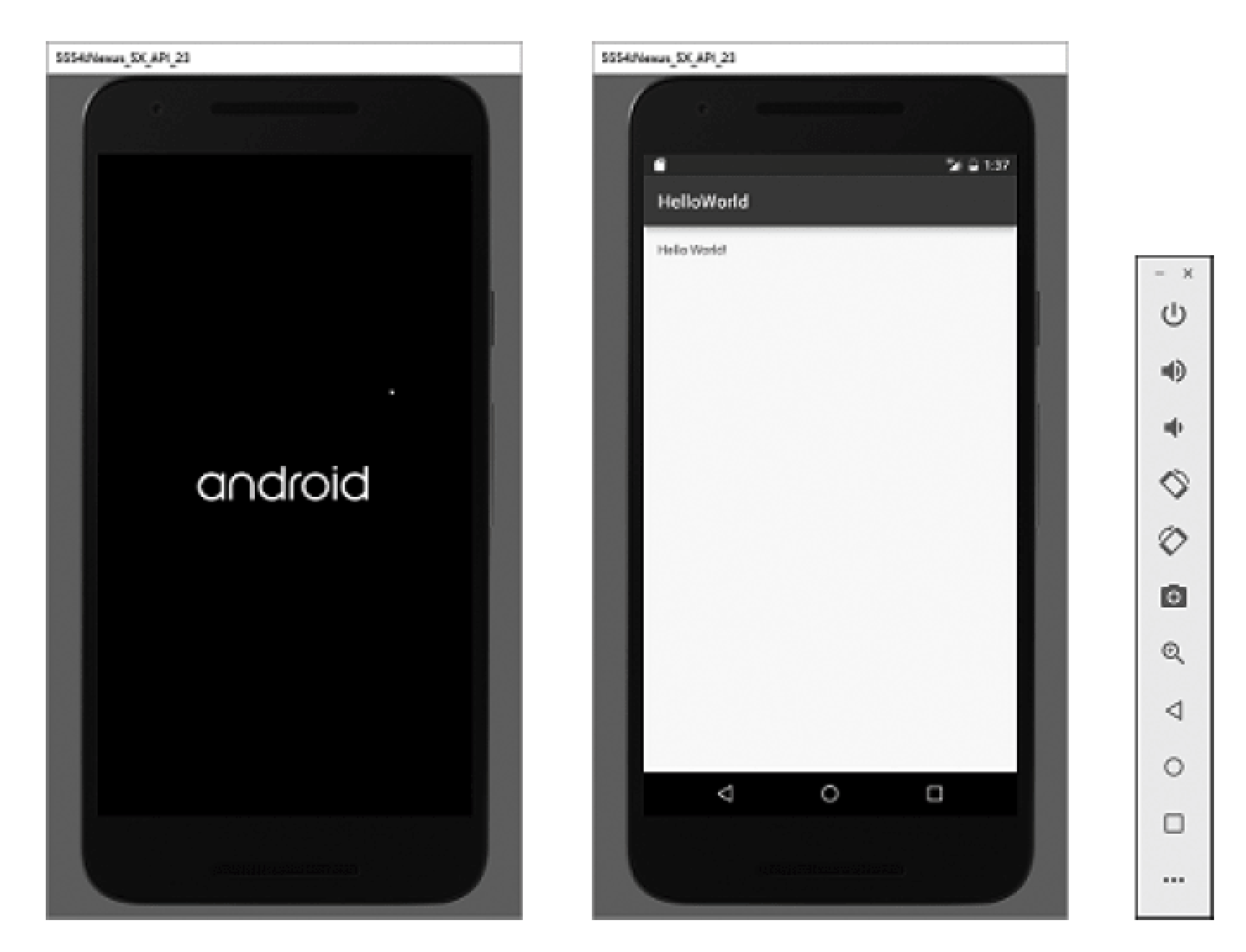


图 1.18　AVD 运行结果

1.3.4　定义简单的用户界面

上面完成的“Hello World!”应用程序，其用户界面所显示的内容是由 Android Studio 的默认布局文件 activity_main.xml 定义。其中使用元素 TextView 定义了一个文本框，来显示“Hello World!”的内容。

Android 系统推荐使用 XML 布局文件来设计用户界面。所有的 XML 布局文件都位于 App 项目的 res/layout/目录中。使用这种方式，可以很方便地定义结构化的用户图形界面，图形组件的布局和相互之间的关系可以很容易、清楚地设定。同时也可以避免直接在源代码中构建应用程序的用户界面，导致一些小的布局变化引起 Java 源代码的大量修改和重新编译。

在 Android Studio 中提供两种定义 XML 布局文件的方式：图形化构建和编写 XML 代码。在这里，采用直接编写 XML 代码的方式，定义一个自己简单的用户界面，替换前面的“Hello World!” 用户界面。

1. 编写布局文件

在 Project 的 app 目录中，展开 res/layout/文件夹，双击打开 activity_main.xml 文件，清除原来的所有内容，替换成代码 1.3 中的内容，编写自己的用户界面布局。

代码 1.3　自己定义的 activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
</LinearLayout>
```

Android App 用户图形界面是使用 Android 系统的 View 和 ViewGroup 子类，按层次结构搭建而成的。例如按钮和文本框等 UI 小组件是 View 对象，而列表和网格等容器是 ViewGroup 对象。Android XML 布局文件就是通过树形结构，来说明在屏幕上所要显示的各组件相互的层次关系和具体如何显示的。一个 Android XML 文件的总体结构比较简单，就是一棵 XML 元素的树，树中的每个节点都使用 View 类的子类名作为元素名称，每一个元素都对应一个用户图形界面中的组件。通过树的根和分支之间的关系，定义了用户图形界面中组件之间的关系。

在代码 1.3 的 XML 布局文件中，就是一个以 LinearLayout 元素作为根的一棵树，在该布局文件中定义了两个图形组件 EditText 和 Button 作为子元素。下面对 acitvity_main. xml 布局文件中的代码含义进行解释。

1）android:id

这个属性为子元素所定义的 View 对象提供了唯一的标识，应用程序可以通过这个 ID 识别、操作或获取这个对象。

另外，在 XML 文件中引用任何一个其他 XML 文件中的资源时，，需要以@开头，后面跟随资源的类型、一个斜线和资源名称。例如，"@string/edit_message"表示应用 string 类型的资源，这个资源的名称是 edit_message。string 资源在/res/strings 中定义。

如果在@后、资源类型前添加一个＋，表示定义一个资源的 ID。

2）android:layout_width 和 android:layout_height

这两个属性定义 View 对象的可用宽度和长度。值 match_parent 表示占据其父节点 ViewGroup 所占据的整个空间；值 wrap_content 表示根据 View 的内容大小来设定对象的大小。通过设定不同的值，View 对象的外形有所变化，执行性能也会有所不同。例如在这里 EditText 的属性中 layout_weight＝"1"和 layout_width＝"0dp" 配合使用，会减少系统计算量，改善应用的性能。

3）android:hint

这个属性设定 EditText 中默认显示的字符串。如果没有预先设定这个属性，EditText 初始显示为空。这里没有直接给这个属性赋一个字符串常量值，而是使用了一个 string 资源引用@string/edit_message，指向字符串资源文件中定义的名称为 edit_message 的资源。字符串定义在 res/values/strings. xml 文件中。这是一种推荐的方式，

因为这可以使应用程序能够很好地本地化，而不需要改变布局文件。

4）android：text

这个属性设置了 Button 上显示的文本。这里同样使用了字符串资源而不是直接给出了字符串值。

2. 添加 string 资源

在 Project 的 app 目录中，展开 res/values/文件夹，双击打开 strings. xml 文件，添加新的字符串定义。一个名称为 edit_message，值为 Enter a message；另一名称为 button_send，值为 Send。代码 1. 4 为添加新的字符串定义后 strings. xml 的内容。

代码 1. 4　strings. xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">My First App</string>
    <string name="edit_message">Enter a message</string>
    <string name="button_send">Send</string>
</resources>
```

3. 运行 App

单击 Run App 运行按钮，运行修改后的 App，可以得到新的界面，见图 1. 19。

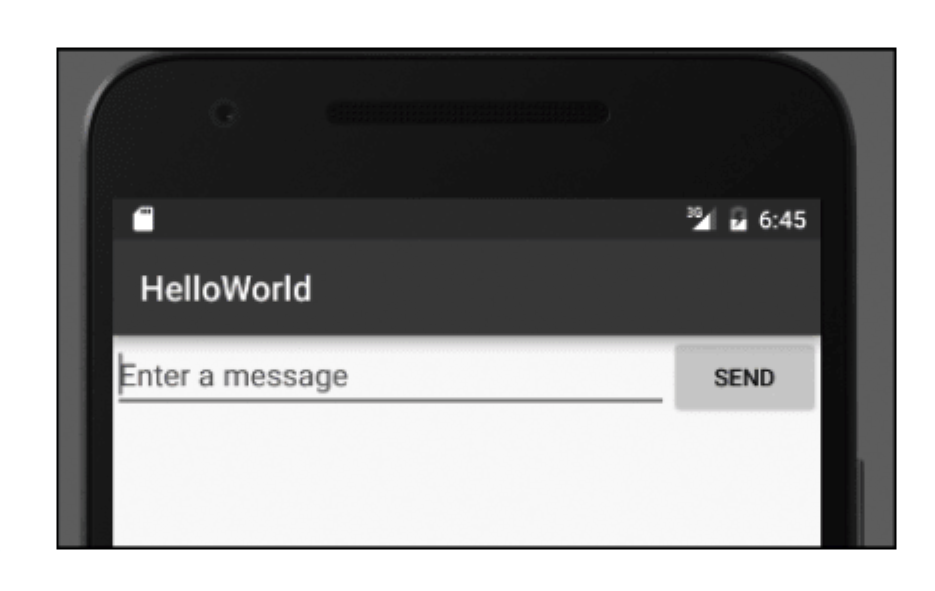


图 1. 19　新的 MainActivity 界面

1.3.5　启动另一个 Activity

完成上面的步骤之后，实现了自己设计的 MainActivity 界面，但界面上的 Button 还不能响应单击的事件。下面实现单击 SEND 按钮后，把 EditText 输入的信息传递到另一个 Activity，通过这个消息启动该 Activity，显示传递过来的信息。具体实现需要下面几个步骤。

1. 响应 SEND 按钮单击事件

首先，修改 MainActivity 的布局文件 res/layout/activity_main. xml，给 Button 元素添加一个属性 android：onClick，见代码 1. 5。

代码 1. 5　添加 Button 的 android：onClick 属性

```
<Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/button_send"
      android:onClick="sendMessage" />
```

android：onClick 的属性值可以设定为某个字符串。设定后，任何时候单击这个按钮，系统就会调用这个图形组件所在的 Activity 子类中以这个属性值命名的方法来进行

事件处理。android:onClick 的值由程序员自己来设定。例如，这里这个按钮的 android:onClick 值设定为 sendMessage()，单击这个按钮时，则由 MainAcivity 中的 sendMessage()方法来进行按钮单击事件处理。该 sendMessage()方法由程序员手动添加。

在添加如代码 1.6 所示的 sendMessage()方法时要注意三点：

- 方法的访问控制权限必须是 public。
- 返回值的类型必须是 void。
- 只能使用一个 View 对象作为参数，这个对象就是产生事件的对象。

代码 1.6　添加 sendMessage()

```
package it.uibe.edu.cn.helloworld;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity {
    public final static String EXTRA_MESSAGE="it.uibe.edu.cn.helloworld.
    MESSAGE";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** Called when the user clicks the Send button */
    public void sendMessage(View view) {
        Intent intent=new Intent(this, DisplayMessageActivity.class);
        EditText editText=(EditText) findViewById(R.id.edit_message);
        String message=editText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
}
```

下一步就可以在 sendMessage()方法中添加事件处理代码，实现信息传递和启动另一个 Activity 了。

2. 创建 Intent 对象传递消息

Android 的 Intent 机制可以在运行时绑定两个独立的组件，利用 Intent 对象在不同组件之间传递消息。

修改 MainActivity.java 文件，在 sendMessage()中创建 Intent 对象，通过 R.id.edit_message 获取布局文件中定义的 EditText 对象，并通过 Intent 的 putExtra()方法，把 EditText 中的文本信息赋给 Intent 对象，然后通过 Intent 启动另一个指定的、名称为 DisplayMessageActivity 的 Activity，见代码 1.7。

代码 1.7　修改后的 MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    public final static String EXTRA_MESSAGE="it.uibe.edu.cn.helloworld.
    MESSAGE";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** Called when the user clicks the Send button * /
    public void sendMessage(View view) {
        Intent intent=new Intent(this, DisplayMessageActivity.class);
        EditText editText=(EditText) findViewById(R.id.edit_message);
        String message=editText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
}
```

注意在 MainActivity 定义的第一行添加 EXTRA_MESSAGE 常量的定义。

修改代码后，平台会通过红色字体提示错误。这时，可以通过连续按 Alt+Enter 键载入所需的类。DisplayMessageActivity 是需要在下一步定义的另一个 Activity。

3. 创建另一个 Activity

在 Android Studio 的 Project 窗口，右击 app 文件夹，选择 New→Activity→Empty Activity，创建另一个 Activity，见图 1.20。

在 Activity 配置窗口的 Activity Name 对应的文本框中输入 DisplayMessageActivity，可以看到默认布局文件名设置为 activity_display_message.xml，然后单击 Finish 按钮。

打开 res/values 目录下的 activity_display_message.xml 文件，添加 TextView 元素，见代码 1.8。

代码 1.8　添加 TextView 组件

```
<TextView android:id="@+id/text_show_msg"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

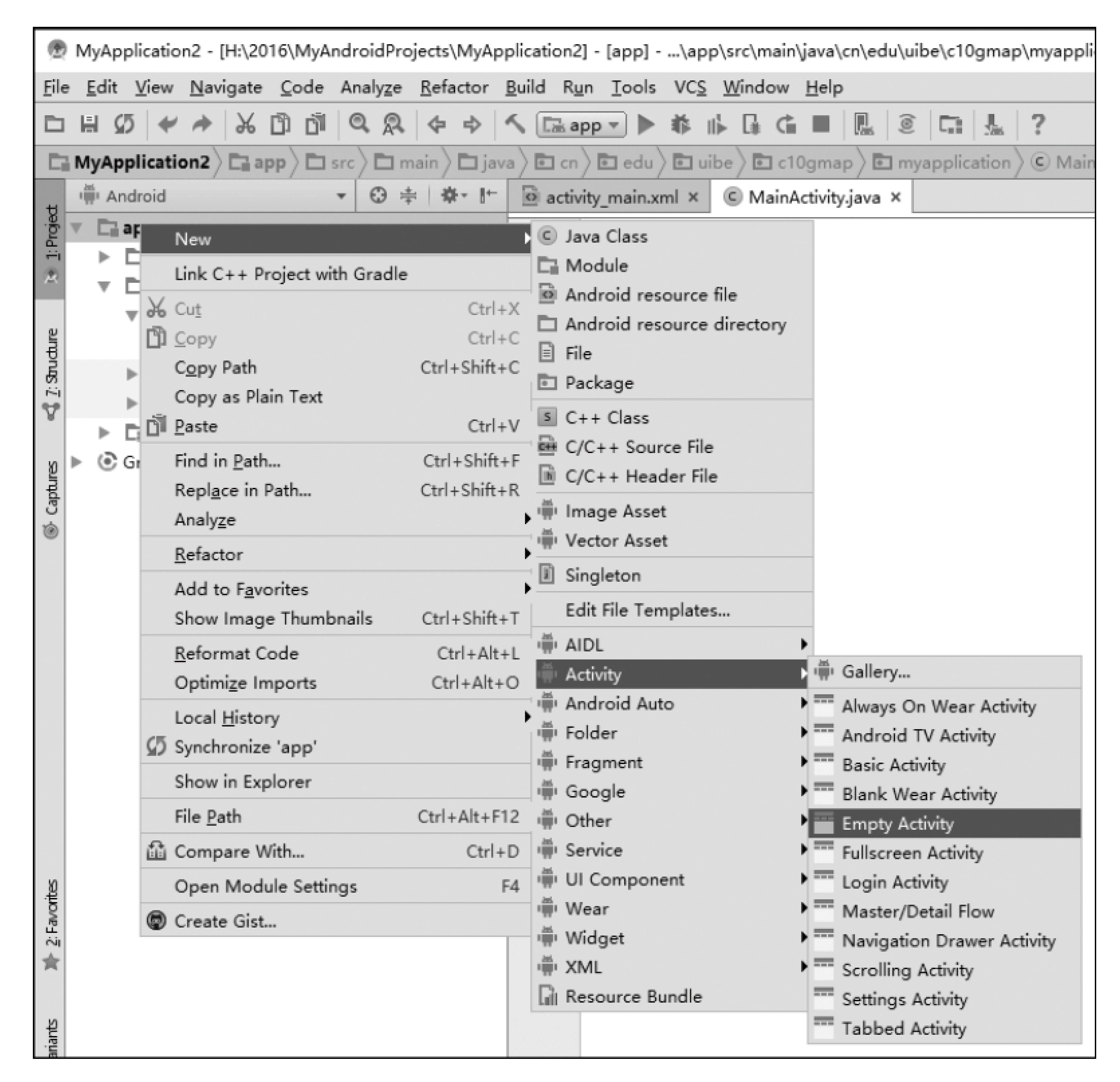


图 1.20　创建新的 Activity

打开 java 目录 it. uibe. edu. cn. helloworld 包的 DisplayMessageActivity. java 文件，在 onCreate()方法中添加获取 MainActivity 发送 Intent 中的信息，并把信息显示在布局文件定义的 TextView 组件上，见代码 1.9。

代码 1.9　DisplayMessage

```
package it.uibe.edu.cn.helloworld;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.TextView;

public class DisplayMessageActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_display_message);

        Intent intent=getIntent();
        String message=intent.getStringExtra(MainActivity.EXTRA_MESSAGE);

        TextView textView=(TextView) findViewById(R.id.text_show_msg);
        textView.setTextSize(40);
        textView.setText(message);
    }
}
```

4. 添加该 Activity 到 manifest 文件

完成 Activity 的代码编写后，需要把这个 Activity 元素添加到 manifest 文件中，打开 manifests 文件夹中的 AndroidManifest.xml，会发现 DisplayMessageActivity 元素已经被 Android Studio 加入到了文件中，见代码 1.10。

代码 1.10　新的 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.uibe.edu.cn.helloworld">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name=".DisplayMessageActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

在以后开发过程中，每一个 Activity 都必须添加到 AndroidManifest. xml 中，如果发现没有自动添加，就必须手动把信息添加进去。Manifest 文件的具体编写，和元素与属性的含义会在下一节详细说明。

5. 运行 App，发送并显示消息

单击 Run App 运行按钮，运行修改后的 App，在 MainActiivty 界面输入信息，单击 SEND 按钮，可以看到系统启动了另一个 Acitivity，即 DisplayMessageActivity，并把前一个 Activity 中输入的信息显示在屏幕上，见图 1. 21。



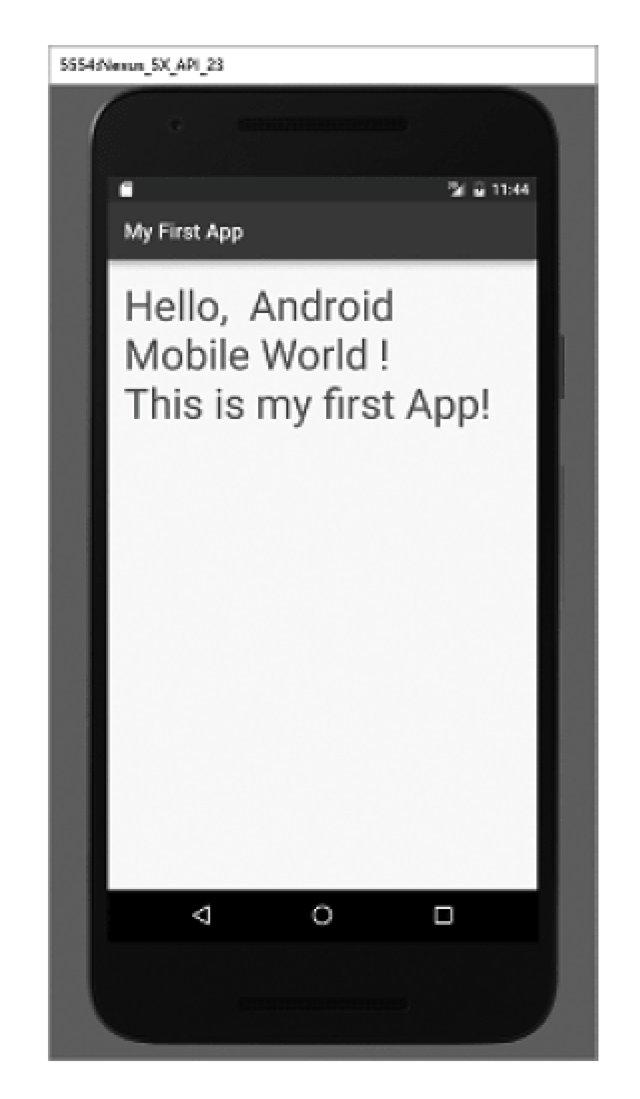


图 1. 21　运行结果

到此为止，就完成了第一个 App 的开发。

1. 4　使用 Android Studio

1.4.1　Android Project 的目录结构

每一个 Android App 都可以由一个 Project 来实现。Android Project 把与源文件和资源文件等项目相关的文件划分为不同的模块进行管理。模块的主要类型如下：

- app 模块(Android App Modules)。
- 库模块(Library Modules)。
- Google App 引擎模块(Google App Engine Modules)。

默认情况下，Android Studio 2. 1. 2 在 Android Project 视图中可以显示当前开发的 Project 的所有文件，见图 1. 22。这些文件分布在不同的模块中，根据其特征很容易访问。

在 Gradle Scripts 模块中包含所有的 build 文件。app 模块包括三个个文件夹：

- manifests：包含 AndroidManifest. xml 文件。
- java：包含 Java 源代码文件，其中包括 JUnit 测试代码。

• res：包含所有的非代码资源，例如 XML 布局文件、UI 字符串和位图图片等。

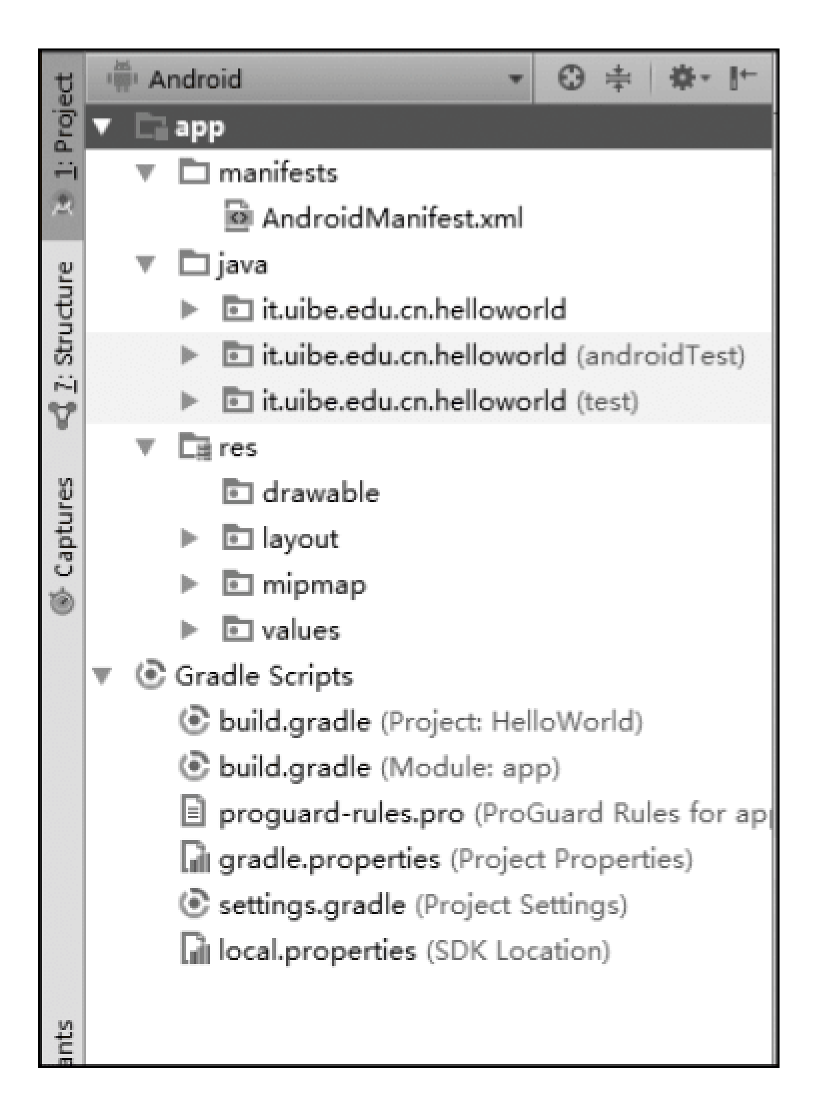


图 1.22　Project 模块和文件

其中，AndroidManifest.xml 主要是说明 App 的名称、版本、SDK，以及 App 项目中有哪些组件、权限如何配置、在一些条件下启动哪些组件。在 App 中运行的所有组件都必须在 AndroidManifest.xml 说明，进行必要的配置，否则无法执行。Android-Manifest.xml具体的设置和所使用的语法，在 1.4.2 节中详细描述。

Java 源代码文件是 App 中的可执行代码，主要是 App 功能执行逻辑的描述以及实现。不同组件和功能的具体实现方式，在后续章节中具体描述。

res 目录为资源目录，而 res 就是 resources 的缩写，分别是 Java 源代码运行中所需要使用的不同资源，这些资源以不同的文件格式分类存储在不同的目录下。

res 目录下所有的资源文件都会在 R.java 文件下生成对应的资源 ID，可以直接通过资源 id 访问到对应的资源；这个 R 文件可以理解为字典，res 下每个资源都会在这里生成一个唯一的 ID。下面介绍一些常用的 res 资源。

1．图片资源 res/ drawable/

res/ drawable/目录用来存放各种位图文件，例如.png、.jpg、.9.png、.gif 等文件，也可能是一些其他的 drawable 类型的 XML 文件。通常位图文件存储的子目录如下：

• mipmap-hdpi：高分辨率，适合目前大多数手机，开发时一般使用这种格式。
• mipmap-mdpi：中等分辨率，很少，适合低端或者比较老的手机。
• mipmap-xhdpi：超高分辨率，手机屏幕材质越来越好，以后估计会慢慢往这里过渡。
• mipmap-xxhdpi：超超高分辨率，适合高端机。

Frame 动画资源也放在这个目录，通过 R.drawable 类来访问。

2．布局文件 res/layout/

res/layout/目录用来存放布局文件。activity_main.xml 布局文件是 Android 项目生成时自动产生的主界面布局文件，程序员可以根据自己的界面设计，定义、命名自己的布局文件，还可以根据不同的手机尺寸或分辨率，为同一个功能定义不同的布局。

3．res/mipmap

该目录定义应用程序使用的图标资源。

4．res/values/

该目录下使用不同文件定义菜单、样式、颜色、维度、字符串、字符串数组和数量字符串等。经常使用的文件如下：

• demens.xml：定义尺寸资源。

- strings. xml：定义字符串资源。
- styles. xml：定义样式资源。
- colors. xml：定义颜色资源。
- arrays. xml：定义数组资源。
- menu. xml：定义菜单资源。
- attrs. xml：自定义控件的属性。
- theme. xml：定义主题文件。

例如，strings. xml 用于定义应用程序或其他资源文件中使用的字符串和数值等，类似于定义属性值的资源文件。前面创建的 HelloWorld 项目中的 strings. xml 文件内容如下：

```
<resources>
    <string name="app_name">HelloWorld</string>
</resources>
```

其中，string 标签声明一个字符串，name 属性指定其引用名。把应用中出现的文字单独存放在 strings. xml 文中的原因有二：一是为了国际化；二是为了减少应用体积，降低数据冗余。

5. 动画资源 res/anim/

res/anim/用于定义预先确定的 Tween 动画资源，通过 R. anim 类来访问。使用 Tween 方式实现的动画，可以使视图组件移动、放大、缩小以及产生透明度的变化，相对于传统按帧播放的 Frame 动画，更具有灵活性。

1.4.2　AndroidManifest.xml 分析

AndroidManifest. xml 文件是当前 Android 项目的功能清单文件，该文件列出了应用中所使用的所有组件。只有在 AndroidManifest. xml 文件中声明了的组件，才能够在项目启动时运行。

前面所建 HelloWorld 项目的 AndroidManifest. xml 文件内容见代码 1. 11。

代码 1. 11　AndroidManifest. xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.uibe.edu.cn.helloworld">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
```

```
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

AndroidManifest.xml 文件中采用了特定的标记，来描述 Android 项目中应用程序或组件的名称、特性等。下面是一些主要标记的说明。

<manifest>：根节点，描述了 Package 中所有的内容。

xmlns:android：包含命名空间的说明，该命名空间使得 Android 中各种标准属性能在文件中使用。

package：声明应用程序包。

android:versionCode：该应用程序版本代号。

android:versionName：该应用程序版本名称。

uses-sdk：该应用程序所使用的 SDK 版本。

<application>：包含 Package 中 Application 级别组件声明的根节点。此元素也可包含 application 的一些全局和默认的属性，如标签、icon、主题、必要的权限等。一个 manifest 中至多包含一个此元素。

android:icon：应用程序图标。

android:label：应用程序名。

activity：是用户打开的一个应用程序的初始页面，大部分被使用到的其他页面也由不同的 activity 所实现。每个 activity 必须有一个<activity>标记对应，无论它给外部使用或是只用于自己的 Package 中。为了支持运行时查找 Activity，可包含一个或多个<intent-filter>元素来描述 Activity 所支持的操作。

android:name：应用程序默认启动的 Activity。

intent-filter：声明了指定的一组组件支持的 Intent 值，从而形成了 IntentFilter。除了能在此元素下指定不同类型的值，属性也能放在这里来描述一个操作所需的唯一标签、icon 和其他信息。

action：组件支持的 Intent Action。

category：组件支持的 Intent category。这里指定了应用程序默认启动的 Activity。

在所有的元素中只有<manifest>和<application>是必需的，且只能出现一次。如果一个元素包含有其他子元素，必须通过子元素的属性来设置其值。处于同一层次的元素的说明是没有顺序的。

在文件的树中，最外层的<manifest>中包含了包名如 package="cn.androidlover.demo"、软件的版本号 android:versionCode="1" 以及 android:versionName="1.0"的属性。

里面一层子元素<application>分支中，将可能包含 Android 程序的四大基础组件 Activity、Service、ContentProvider 以及 Receiver 中任一类型的对象。如果在应用程序中添加上面四个类型中的任一种新对象，都需要在 AndroidManifest.xml 文件中添加相应节点，否则运行时将会产生异常。例如，对于一个 Activity 来说，无论给外部使用或是只用于自己的 package 中，都必须要一个<activity>标记对应进行说明。如果 Activity 没有对应的标记，就不能运行。

AndroidManifest.xml 文件中各元素以及它们的属性都是可选的，但即使没有在文件中显示设置，它们也都有默认的设置。除了根元素<manifest>的属性，所有其他元素属性的名字都是以"android:"作为前缀的。

下面是定义 AndroidManifest.xml 元素时需要注意的地方。

- 定义类名。

所有的元素名都对应其在 SDK 中的类名，如果是自定义类名，必须包含类的数据包名，如果类与 application 处于同一数据包中，可以直接简写为"."。

- 多数值项。

如果某个元素的属性有超过一个数值，必须通过重复的方式来说明这个属性具有多个数值项，且不能将多个数值项一次性说明在一个属性中。

- 资源项说明。

当需要引用某个资源时，需要按照规范的格式：@[package:]type:name。例如<activity android:icon="@drawable/icon" …>。

- 字符串值。

类似于其他语言，如果字符中包含有字符"\"，则必须使用转义字符"\\"。

AndroidManifest.xml 文件中的元素是规定的，不能加入自己创建的元素和属性。下面是按照字母顺序排列的所有可以出现在 AndroidManifest.xml 文件里的元素，它们是唯一合法的元素。

```
<action>
<activity>
<activity-alias>
<application>
<category>
<data>
<grant-uri-permission>
<instrumentation>
<intent-filter>
<manifest>
<meta-data>
```

```
<permission>
<permission-group>
<permission-tree>
<provider>
<receiver>
<service>
<uses-configuration>
<uses-library>
<uses-permission>
<uses-sdk>
```

1.5 Android App 开发起步

1.5.1 App 开发流程

Android 应用程序，即 Android App，采用 Java 语言作为开发语言，通过 Android SDK 工具编译代码后，与应用文件所使用的资源文件和数据等部分一起，生成 Android Package 文件，即. apk 文件。当用户下载安装一个 Android App 时，安装的文件就是. apk 文件。

Android App 开发过程中，把应用程序逻辑和用户界面设计完全分隔开，可以分别进行独立的设计和开发。应用程序逻辑使用 Java 代码实现；而用户界面设计可以使用图形工具或 XML 文件，定义图形界面的组件和具体布局。理论上，在这种模式下应用程序的逻辑改变，不会影响用户界面的设计，不需要修改；用户界面设计的布局调整，也不需要修改相应的 Java 代码。虽然实际开发中不能完全实现二者隔离，但大大减少了代码的修改工作量。

在 Android Studio 平台上开发 Android App，与在其他平台上开发的流程基本一致。作为 Android App 专用开发平台，Android Studio 平台把开发过程中所需的一些设计和开发的专用工具，例如用户界面图形设计工具、Android SDK、AVD 等，统一规整到一个安装包中，安装后就具有开发中所需要的所有工具，不再需要逐个安装测试。下面是在 Android Studio 中开发 App 的开发流程，见图 1.23。

1. 安装

在这个阶段，需要下载 Android Studio 安装包，安装 Android 开发环境后，配置 AVD 和测试设备，并通过 HelloWorld App 测试整个平台的各项功能，熟悉所安装版本平台的工具和 Project 的组织结构，为正式开发 App 做好准备。

2. 开发

这个阶段，首先需要创建 App 对应的项目(Project)。所开发 App 的所有相关文件会以 Project 的目录形式分类存放在 Project 中。这一阶段的任务分为三个部分，见图 1.23。

其中，UI 设计主要是用户界面的组件定义和布局设计，可以使用 Android Studio 的

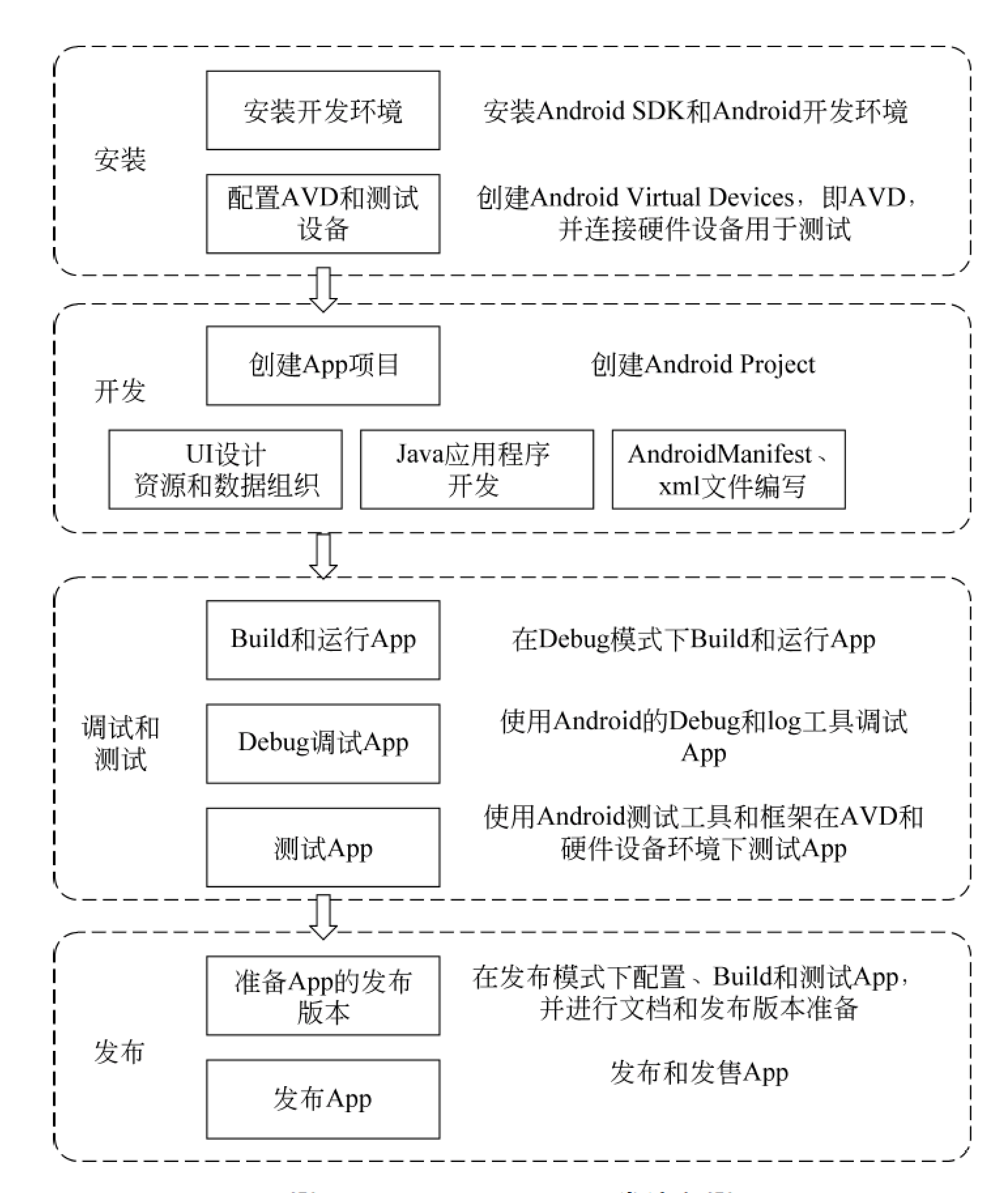


图 1.23　Android App 开发流程图

图形工具实现，也可以直接使用 XML 语言来直接编写。无论哪种形式，最后 UI 的设计代码都以 xml 文件形式存储在资源文件目录 res 中；资源和数据组织主要是把图形、图标、网络链接等资源和数据通过资源文件存放在 res 合适的目录中。

Java 源程序代码，主要实现用户界面的交互功能和后台的数据管理、网络通信等程序功能。通过 Java 源程序代码，把 UI 设计的用户界面按照应用逻辑关联起来，实现整个 App 的完整功能。在开发阶段，需要进行每个 Java 源程序的编写、调试和运行。

AndroidManifest.xml 文件是对 App 总体进行配置的文件，在 Java 应用程序开发的过程中，每一个 Android 组件都需要在这个文件中进行配置后，才能够添加到 App 中执行。AndroidManifest.xml 中还需要进行其他的一些设置，具体的内容可以参照 1.4.2 节。

3. 调试和测试

这个阶段，其中一个重点是配置 Build 的模式，进行软件测试，即是对 App 进行总体 Build、调试和测试，可以按照软件的测试标准，书写测试用例，查找软件的中 bug，对软件的功能和性能进行测试，优化 App。

另一个重点是在不同配置的 AVD 上和硬件设备上对 App 进行功能和性能测试，对

其兼容性和健壮性进行测试，调整 App 的用户界面友好程度和增强 App 的版本兼容性。

这些调试和测试过程，Android Studio 都提供有相应工具来帮助程序员进行，具体的使用可以参考 Android Studio 的手册。

4. 发布

发布是把开发完成的 App 提供给用户的过程。这个过程需要完成两个任务：准备发布的 App 和在网络上发布 App。

1.5.2 Android 关键组件

Android App 都是由 Android 组件搭建而成的。在众多的组件中，关键的基本组件有四种：Activity、Service、ContentProvider、BroadcastReceivers。Android App 根据自身的功能需求，使用其中一种或多种组件。在这里简单介绍这些组件，对其功能有所了解。

在有的资料或书籍中，把这些组件的英文名称翻译成相应的中文名称，但因为在程序设计和源程序代码编写中，都是直接使用这些组件的子类，所以本书中在提到这些组件时，直接使用英文名称，方便大家理解和记忆，不再进行转换。

Activity 是一种关键的执行组件，为用户操作而展示的可视化用户界面。在 App 中至少需要一个 Activity 作为 UI 界面，所有的 Activity 都是从 Activity 类继承而来。对于 App 来说，Activity 是程序的入口，相当于 C/C++ 中的 main()。例如，照相机 App 可能有一个 Activity 显示一个聚焦或拍照的可视化界面，也能启动另一个 Activity 来管理和像是所拍的照片。App 中的每一个 Activity 都是独立的，但可以相互转换。

Service 主要用于实现一种需要长时间执行、不需要用户界面的后台操作。所有的 Service 都是 Service 的子类，可以由另一个的应用组件启动。

ContentProvider 是 App 的共享数据处理组件，提供自己的数据给外部应用程序使用，提供的数据可以存储为 Android 文件、SQLite 数据库文件或其他合法格式。它主要应用于不同的应用程序之间进行数据共享，如果某一应用程序需要使用其他应用程序的数据，就必须采用 ContentProvider 对象。

在 Android 的系统中系统会发送的广播通知，例如电池电量过低或者信号过低，其他 Android 组件也会发送广播消息。BroadcastReceiver 是负责接收广播消息并对消息做出反应的组件。如果在一个 App 中需要接收某种广播消息并进行处理，就可以定义一个 BroadcastReceiver 的子类。BroadcastReceiver 不会与 UI 交互。但一般来说，当广播事件发生时，会创建一个状态条通知来提醒用户。

1.6 本章小结

本章首先简单介绍了 Android 的概念和 Android 系统的技术架构；然后详细介绍了 Android 开发环境 Android Studio 在不同操作系统中安装和配置的具体步骤，使用 HelloWorld App 说明了在 Android Studio 中 App 开发和运行过程，着重介绍了 App 在硬件设备和 AVD 上运行时配置的过程；在最后两节，简单介绍了 Android Studio 的使用和 Android App 的开发流程。

第 2 章 设计应用界面

每个 Android 客户端应用首先面对的就是界面的开发。Android 系统提供了丰富的界面控件。Android 提供的用户图形交互界面称为 Activity。本章主要介绍 Activity 的基本知识和使用方法,如何在 Activity 上实现不同的布局,如何理解字符串、图片等资源的使用。

2.1 理解 Activity

Activity 是 Android 的四大基本组件之一。通过 Activity,用户可以与移动终端进行交互,使用 Android 应用程序做一些事情,如拨打电话、拍照、发送电子邮件或查看地图等。Activity 也可以看作是一个特定的窗口,输入框、按钮等各种视图控件能够在其中按需求进行不同的排列。这种窗口通常填满整个屏幕,但可能会小于屏幕或者浮在其他窗口之上,例如对话框。

所有的 Activity 都是从 Android 提供的类 Activity 继承而来。一个应用程序通常由一个或多个 Activity 组成。

在应用程序中,通常指定一个主 Activity,它是应用程序启动时,首先呈现给用户的界面。App 通过主 Activity,根据不同功能启动其他的 Activity。这些 Activity 之间根据应用程序的逻辑功能可以实现相互调用启动。如果一个 Activity 启动了另一个 Activity,则自身的状态发生改变,处于 stopped(停止)状态,新的 Activity 接替它成为用户可操作的界面。通常情况下,一个 App 不会将所有的功能在一个 Activity 中实现。

在第 1 章编写第一个 Android 应用程序显示"Hello World!"时,用到了 Activity。下面来讨论创建和使用 Activity 的基本过程,了解 Activity 在用户操作过程中呈现的状态和各种状态的转换过程,以及对应的系统回调方法。

2.1.1 创建可运行的 Activity

要创建在 Android 应用程序中可运行的 Activity,必须要实现四个任务:

(1) 命名并定义 Activity 的子类。

(2) 设计和实现用户界面。

(3) 在 Manifest 文件中声明这个 Activity。

(4) 测试运行。

1. 命名并定义 Activity 的子类

要创建一个用户界面 Activity，必须定义一个子类来继承 Activity，并根据界面的功能和状态变化，在其回调方法中编写相应的代码。

在 Android 定义 Activity 组件时，针对用户界面的不同状态变化，为其定义了一系列不同的回调方法。例如用户界面创建、改变、恢复或销毁的动作，都对应 Activity 中不同的回调方法。所谓回调方法，就是在 Activity 状态变化时，Android 系统会自动调用在 Activity 中预先定义的对应方法，执行其中的代码。因为是系统反向调用子类中定义方法，实现其功能，所以称为回调。因此定义子类时，可以在回调方法中定义需要的功能，系统就可以在用户界面状态变化时，调用应用程序中 Activity 的对应回调方法，来实现其功能。例如，某个 Activity 需要在界面恢复时重新读取数据库的信息，就可以在此 Activity 的 onResume()回调方法中定义重新读取数据库的操作。这样，当这个 Activity 界面恢复时，系统就会自动调用它的 onResume()，运行里面定义的代码，实现其功能。

在 Activity 的回调方法中，有两个最重要的回调方法 onCreate()和 onPause()。

1）onCreate()

在系统创建 Activity 时，第一个调用这个方法，并且执行其中的代码，因此必须实现这个方法。这部分代码主要是进行变量的初始化，完成 Activity 的初始化，其中最主要的是必须调用 setContentView()方法，为 Activity 的用户界面定义布局，就是初始化显示界面。

一旦 onCreate()方法调用完成后，Activity 的状态不会停留在 Created，系统会立刻接着调用 Acitivity 的另两个回调方法 onStart()方法和 onResume() 方法，Activity 的状态很快会进入到 Active 状态，也就是运行状态。

2）onPause()

当用户离开当前的 Activiy 时，系统调用此方法。

下面为了讨论 Activity 的创建、使用和状态，在 Android Studio 建立一个新的 App 项目，命名为 C02ActivityLifecycle，主 Activity 命名为 MainActivity，Company Domain 定义为 uibe. edu. cn，并按照 1. 3. 5 节中创建 Activity 的步骤，创建另外两个类 DialogActivity 和 ImageActivity，注意将它们的父类都改为 Activity。在这个 App 中将实现从 MainActivity 分别到 DialogActivity 和 ImageActivity 两个界面的转换和返回。

在完成 Activity 初步创建之后，就进行下一步，根据 App 的功能设计和实现用户界面。

2. 设计和实现用户界面

Android 为程序员提供了三种用户界面设计的方法：Java 代码“编程式”设计实现、XML 文件编写设计和图形化界面设计。

如果使用 Java 代码直接在源代码中构建应用程序的用户界面，在编写代码过程中必须对界面的层次关系十分清晰，并逐个定义这些用户图形界面组件的属性和关系，这在界面比较复杂、组件比较多时，对程序员是一个考验。在界面美观、组件排列合理、组件位置微调、风格统一等细节方面可能会耗费程序员较多的时间，可能比预计的时间多得多。这种方式对于界面的开发是很不方便的，因为一些小的布局变化都有可能导致对源代码的

修改，并且需要重新编译。因此这种方法，在 Android 的用户图形界面设计中并不推荐。

设计实现用户界面的第二种方法，XML 文件编写设计，是使用 Android Studio 的 XML 文件编辑器打开 Activity 对应的 XML 布局文件，在文件中编写代码，来描述界面的布局、控件和相关的属性。在 1.3.4 节中使用的就是这种方法。由于 XML 文件是一个层次化结构的文件，因此布局和控件的关系十分清楚，语法比较简单。而且组件的标识、属性和位置的设置集中在一起，布局和控件不涉及应用程序的逻辑，只是单纯的排列和设计，也就能很方便、快速地完成用户界面的布局，修改起来也很容易定位。缺点是在界面比较复杂、组件比较多时也很难兼顾美观，如果只能在 Activity 运行时才能看到效果，调整起来同样也很耗费时间。

图形化界面设计是使用 Android Studio 提供的图形化界面设计工具，能够把工具提供的布局和控件直接拖曳到手机界面上合适的位置，直观地显示出来，并且通过属性设置界面，直接设置组件的属性。在设计过程中，随着布局和控件的添加、属性的修改，XML 布局文件的代码也自动生成。使用第二种方法直接编写代码的 XML 布局文件，也可以通过图形化界面设计的手机的模拟界面，直观地看到实际显示效果。

如果要使用 Android Studio 的图形化界面设计工具，首先打开需要设计的 XML 布局文件，然后单击图 2.1 左下角箭头指向标注的 Design 标签，就可以看到手机模拟界面的窗口了。下面以 MainActivity 的界面设计为例，详细描述如何使用这个工具进行界面设计。

1）打开 XML 布局文件的 Design 界面

在这个例子中，首先双击打开 MainActivity 对应的布局文件 activity_main. xml，单击 Design 标签，进入图形化设计。然后在手机模拟界面上，选中已有的 Hello World！文本框，删除这个组件，准备好空白屏幕，为添加自己的布局和控件做准备。

2）从 Palette 拖曳组件到手机屏幕

图 2.1 中左上角箭头指向标注的 Palette 窗口中，列出了 Layouts、Widgets、TextFields 和 Containers 等布局和控件。在 Palette 窗口中，拖动 Widgets 中的 Large Text 放置到手机屏幕的顶部。在组件放置过程中，会有位置线辅助。这时可以在图 2.1 中右上角箭头指向标注的 ComponentTree 窗口中，看到 Device Screen 的分支 RelativeLayout 的下面出现了名为 TextView 的控件。

接下来，拖动 Containers 中的 ScrollView 放置到屏幕中 Large Text 的下方；再拖动 Widgets 中的 Medium Text 放置到 ComponentTree 窗口的 ScrollView 组件上，让这个 TextView 成为 ScrollVeiw 组件的分支。

3）修改组件的属性

图 2.1 中右下角箭头指向标注的 Properties 窗口，是布局和控件的属性设置窗口。在这个窗口中列出了布局和控件所能设置的所有属性。单击属性对应的一栏，就可以根据设计给予不同的赋值或选择。通常情况下，id、text、layoutwidth、layoutheight、background、onClick 等属性是经常需要设置的属性。

例如，在这里 ScrollView 的 layoutwidth 设置成 fill_parent，layoutheight 设置成 200dp，使其宽度和屏幕宽度一致，高度设置成 200dp，容纳其分支 TextView 显示的内

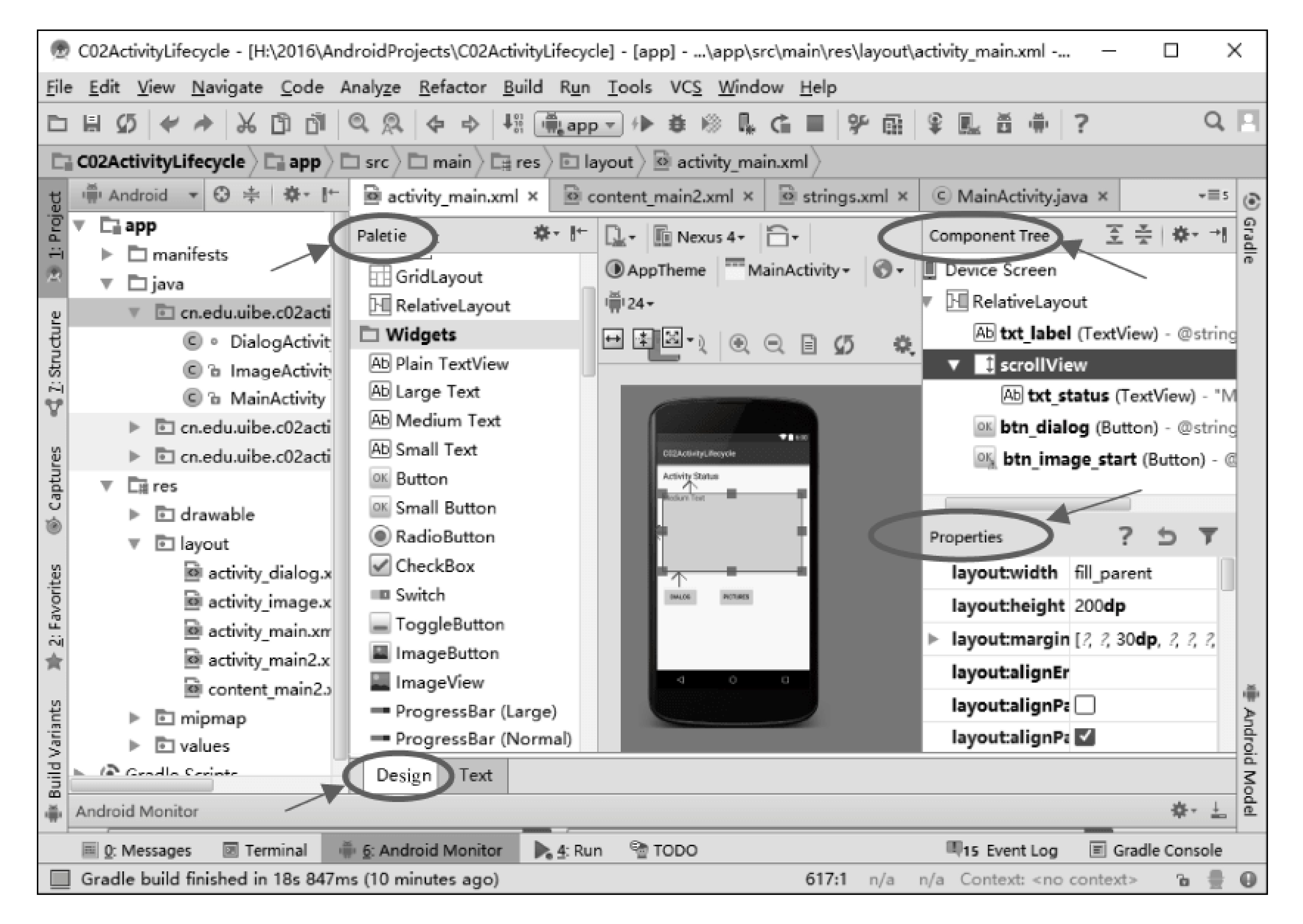


图 2.1　用户界面图形设计工具

容。为了突出显示的信息，其 background 设置成 highlighted_text_material_dark。

继续拖动 Widgets 中的 Button，为屏幕中的 ScrollView 添加两个 Button。修改前一个 Button 的属性，id 为 btn_dialog，text 为 Dailog，onClick 设定其单击事件处理的方法为 startDialog；修改后一个 Button 的属性，id 为 btn_image，text 为 Picture，onClick 设定其单击事件处理的方法为 startImage。

图形化设计完成后，单击 Design 旁边的 Text 标签，可以查看对应的 XML 文件的文本形式代码，对一些细节进行调整和补充。单击 Design 标签和 Text 标签可以使布局文件在图形形式和文本代码形式之间转换。综合使用图形化设计和 XML 代码编写，是一种效率相对较高的设计方式。

4）对布局和控件的事件进行处理

根据上面对 onClick 属性的设置，在 MainActivity.java 代码中添加按钮的事件处理代码（见代码 2.1），实现启动另一个对应的 Activity。

代码 2.1　启动另一个 Actiivity

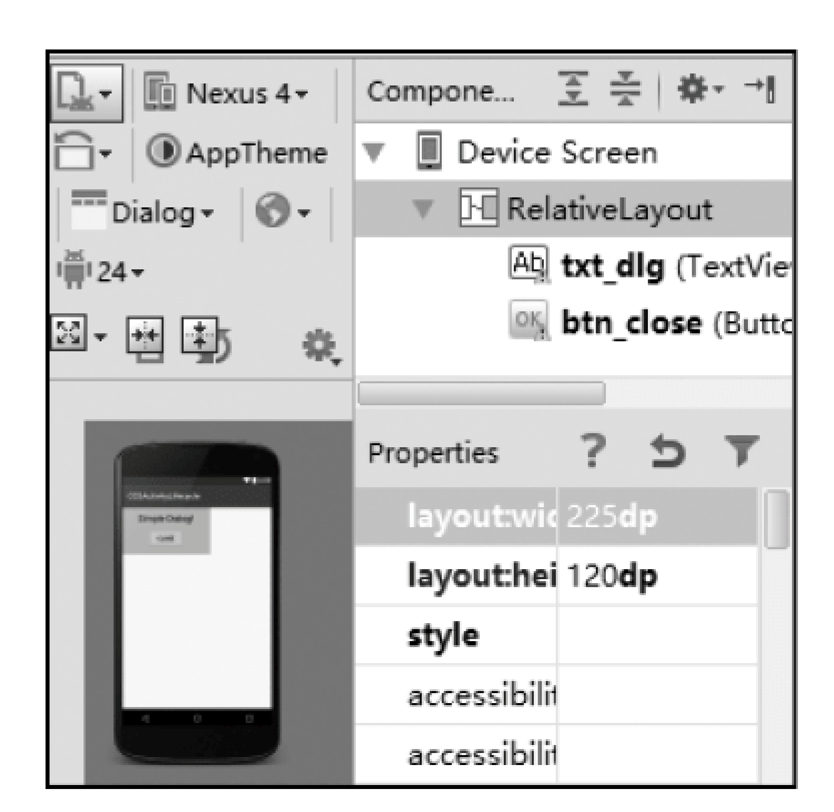
```
//处理单击 Dialog 按钮事件,启动 DialogActivity
public void startDialog(View v) {
    Intent intent=new Intent(MainActivity.this, DialogActivity.class);
    startActivity(intent);
}

//处理单击 Pictures 按钮事件,启动 ImageActivity
public void startImage(View v) {
```

```
    Intent intent=new Intent(MainActivity.this, ImageActivity.class);
    startActivity(intent);
}
```

到此，MainActivity 的设计基本完成。在图形化设计过程中，也可以通过单击组件提示完成 strings. xml 的设置，使用喜欢的图片作为整个 Activity 的背景，或尝试改变其他的属性，实现自己的个性化设计。

另外，通过 MainActivity 启动的 DialogActivity 和 ImageActivity 界面的设计如图 2.2 所示。左图是 activity_dialog. xml 的图形化设计结果，其中 RelativeLayout 的 layoutwidth 值为 225dp，layoutheight 值为 120dp，背景 background 设置为 holo_orange_light，按钮 btn_close 的 onClick 属性设定其单击事件处理的方法为 closeDialog；右图是 actiivty_image. xml 的图形化设计结果，其中 ImageView 对象的 background 使用了 res/drawables 中的自定义图片，按钮 btn_image_close 的 onClick 属性设定其单击事件处理的方法为 closeImage()。



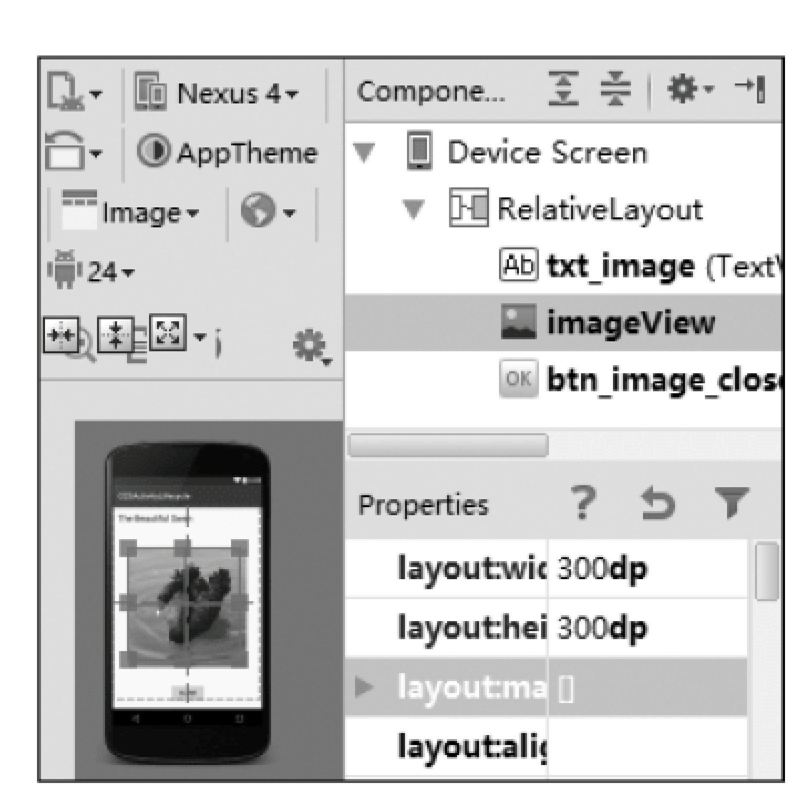


图 2.2　DialogActivity 和 ImageActivty 界面设计

这两个 Activity 的事件处理很简单，也非常类似，都是关闭自身的 Activity。具体代码可以参考代码 2.2 中的 closeDialog()方法。

代码 2.2　DialogActivity. java

```
public class DialogActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_dialog);
    }

    public void closeDialog(View v) {
```

```
        DialogActivity.this.finish();
    }
}
```

3. 在 Manifest 文件中声明 Activity

每个 Android 应用程序都是一个独立的 Android 项目,都有一个 AndroidManifest.xml 文件,这里面是这个项目中所包含的组件和应用程序的配置说明。在创建项目时,Android Studio 会自动创建这个文件。将新定义的 Activity 的相关参数写入 AndroidManifest.xml 文件的过程,称为 Activity 的声明。

Activity 只有在 AndroidManifest.xml 中声明后,才能够在应用程序调用时成功运行,系统才可以访问到它们。

要声明前面新创建的 DialogActivity 和 ImageActivity,使它们可以在 App 启动后正常执行,可以打开 Manifest 目录下的 AndroidManifest.xml 文件,添加两个<activity>元素作为<application>元素的子元素,注意将 DialogActivity 的主题设置成 Dialog,见代码 2.3。

代码 2.3　添加新增的 Activity 到 Manifest 文件

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="cn.edu.uibe.c02activitylifecycle">

    <application
            …
        <activity android:name=".MainActivity">
            …
        </activity>

        <activity
            android:name=".DialogActivity"
            android:theme="@android:style/Theme.Dialog" />
        <activity android:name=".ImageActivity" />
    </application>

</manifest>
```

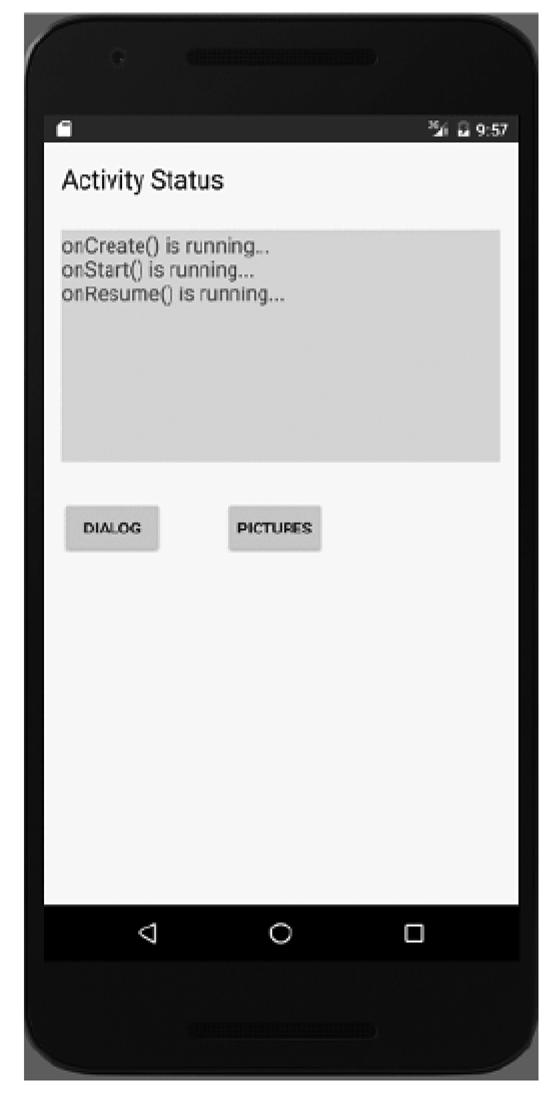
4. 运行 App 测试 Activity 功能

完成上面的步骤后,运行 App,选择 AVD,等待一会儿,就会看到图 2.3 中图屏幕显示出 MainActivity 的界面效果。单击 DIALOG 按钮,会启动 DialogActivity 显示出左图的屏幕,单击 PICTURES 按钮,会启动 ImageActiivty 显示出右图的屏幕。

由于 MainActivity 中还没有添加显示状态转换的信息,在中图屏幕上的有色显示部分还只能看到默认的 TextView 文本。当完成下一节 Activity 的生命周期的例子后,就

可以看到图示的信息了。





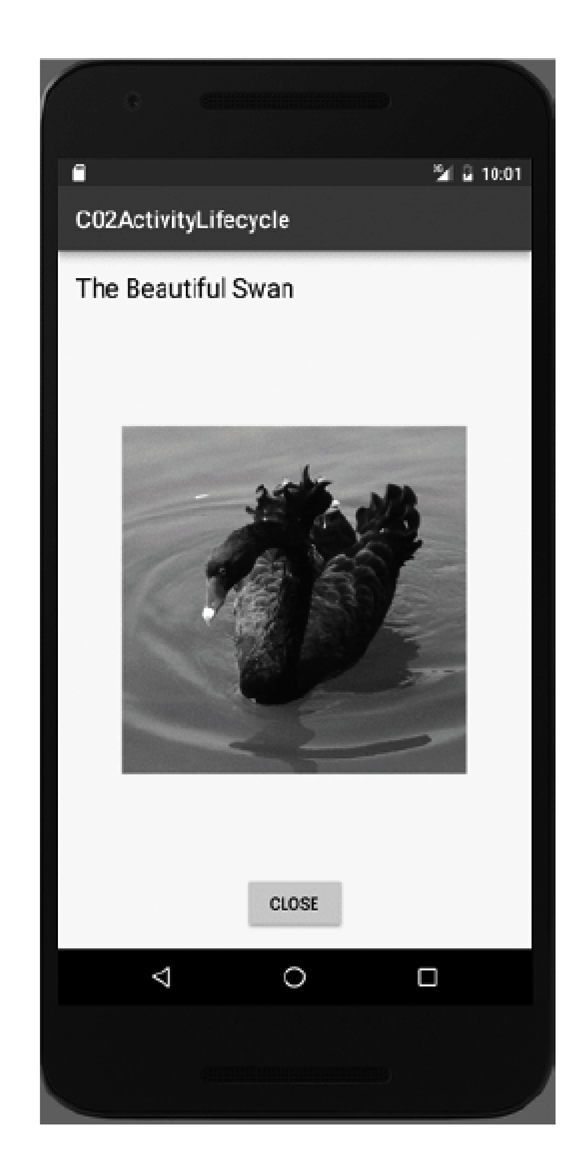


图 2.3　App 中的 Activity 显示效果

2.1.2　Activity 的生命周期

Android 平台主要是为移动终端开发的操作系统。移动终端的特性就是应该能随时在未完成当前任务的时候，切换到其他任务中，再次回来以后还可以继续完成刚才没有完成的任务。为什么设置 Activity 的生命周期呢？

先看一个典型的例子：在编写短信时，有一个紧急电话打过来，你必须要接这个电话，如果接完电话后，你肯定希望继续编辑刚才的短信，完成这个任务。

为了完成类似的任务，Android 系统需要同时执行多个程序。但对于移动终端这种有限资源的平台来说，同时执行多个程序可以提高用户的友好性，但是也有它的严重的缺点。每多执行一个应用程序，就会多耗费一些系统内存。手机里的内存是相当有限的，当同时执行的程序过多，或是关闭的程序没有正确释放掉内存，执行系统时就会觉得越来越慢，甚至不稳定。为了解决个问题，Android 引入了生命周期的机制来管理应用程序，见图 2.4。

Android 应用程序的生命周期是由系统框架进行管理，不是由应用程序直接控制，Android 系统的 Dalvik 虚拟机会依照内存状况和 Activity 的使用状态，来自动管理内存的使用。

图 2.4 说明了 Activity 生命周期中各个状态，以及在状态转换过程中系统会调用的方法。Activity 状态的转换是由于用户的操作或其他原因引起的，当 Activity 状态发生转换时，相关方法中的代码就会执行。例如在 Activity 创建启动时，会依次调用其 onCreate()、onStart()、onResume()方法中的代码；当 Activity 从运行状态转换成暂停状态时，会调用其 onPause()方法中的代码。

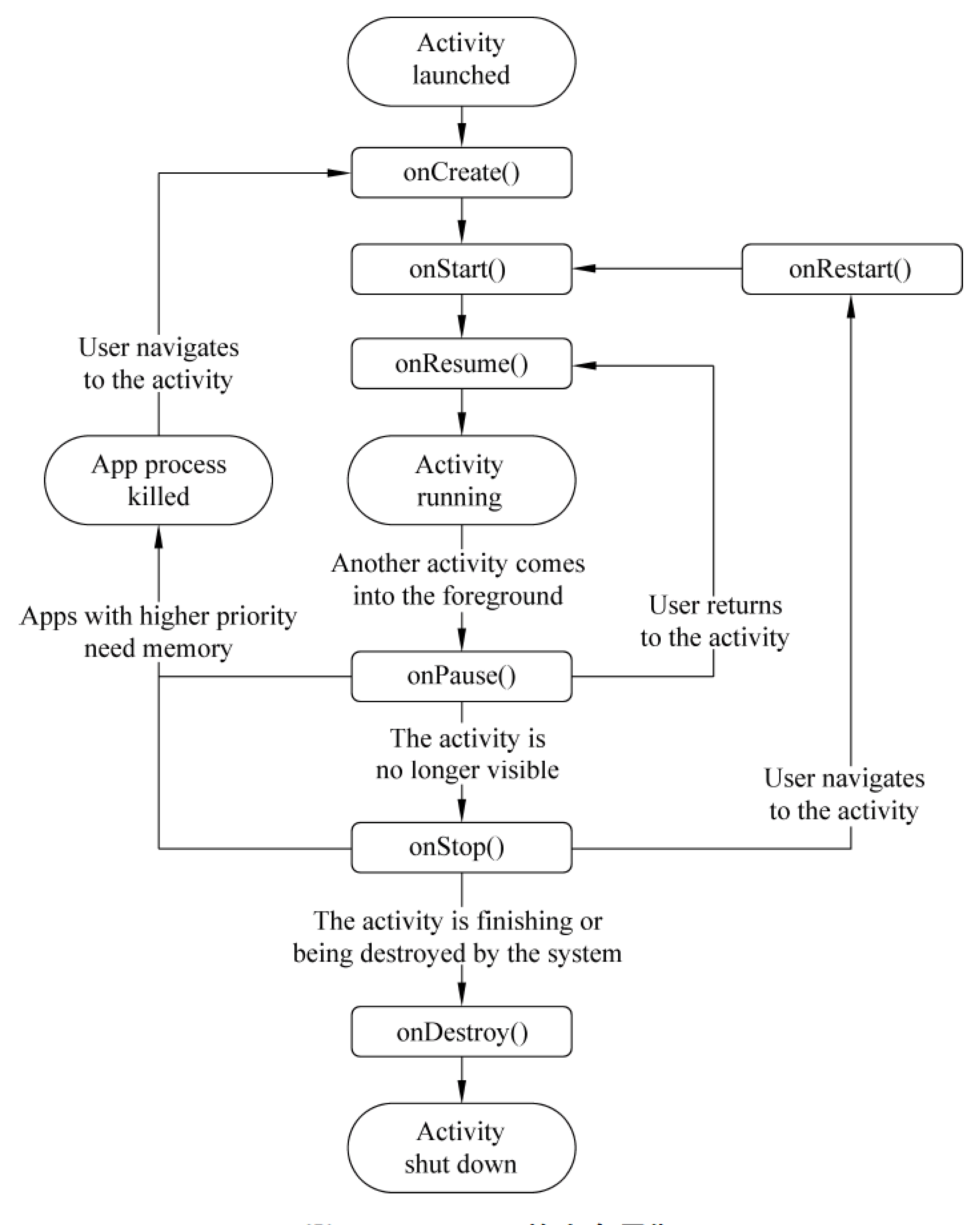


图 2.4 Activity 的生命周期

Activity 的生命周期中有四种状态：

- Active/Running：在屏幕的前台，叫做活动状态或者运行状态（Active or Running）。
- Paused：如果一个 Activity 失去焦点，但是依然可见（一个新的非全屏的 Activity 或者一个透明的 Activity 被放置在栈顶），叫做暂停状态（Paused）。一个暂停状态的 Activity 依然保持活力（保持所有的状态、成员信息，和窗口管理器保持连接），但是在系统内存极端低下的时候将被杀掉。
- Stopped：如果一个 Activity 被另外的 Activity 完全覆盖掉，叫做停止状态（Stopped）。它依然保持所有状态和成员信息，但是它不再可见，所以它的窗口被隐藏，当系统内存需要被用在其他地方的时候，Stopped 的 Activity 将被杀掉。
- Killed：如果一个 Activity 是 Paused 或者 Stopped 状态，系统可以将该 Activity 从内存中删除，Android 系统采用两种方式进行删除：要么要求该 Activity 结束，要么直接杀掉它的进程。当该 Activity 再次显示给用户时，它必须重新开始和重置前面的状态。

Activity 的生命周期中有三个关键的循环，见图 2.5。

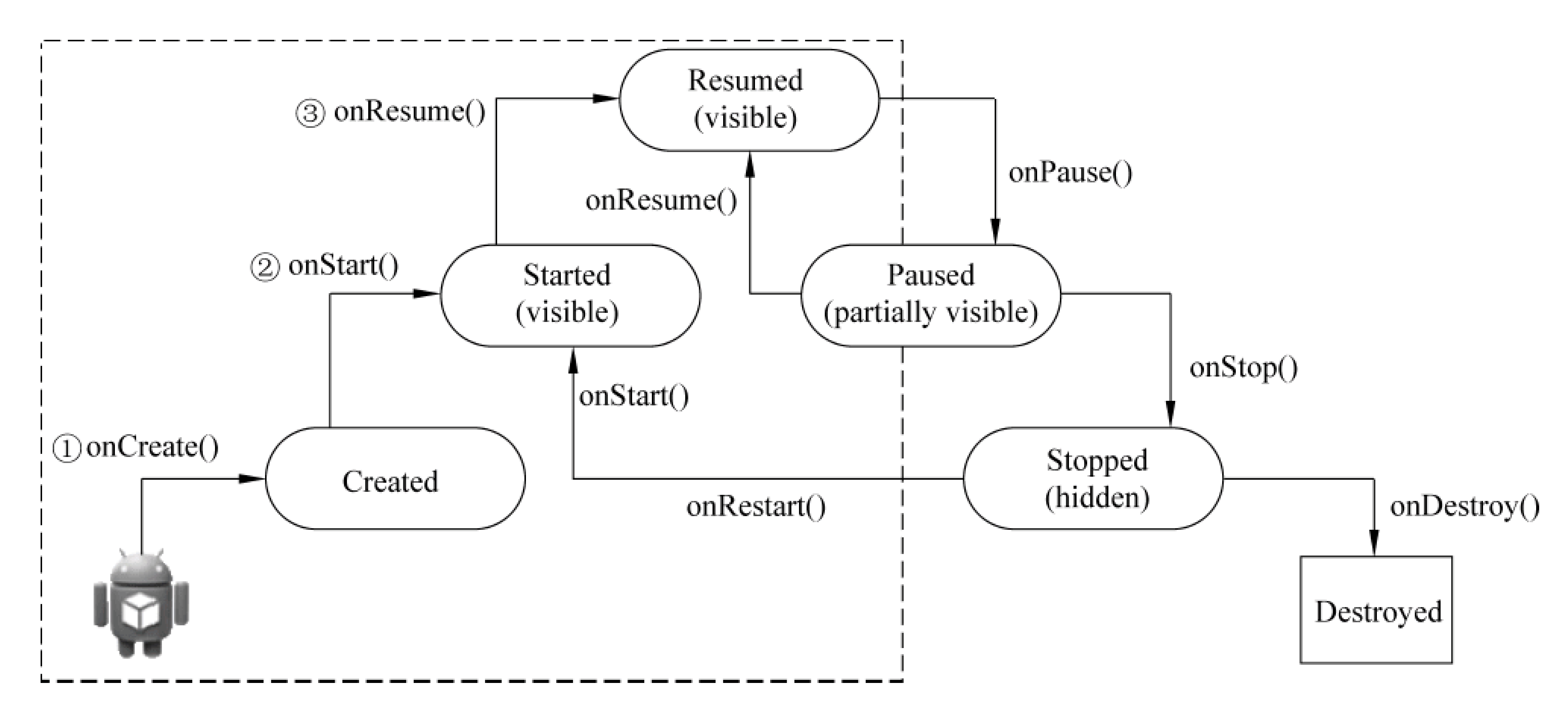


图 2.5　Activity 的状态转换

- 整个的生命周期。从 onCreate 开始到 onDestroy()结束。Activity 在 onCreate()设置所有的"全局"状态，在 onDestory()释放所有的资源。例如：某个 Activity 有一个在后台运行的线程，用于从网络下载数据，则该 Activity 可以在 onCreate()中创建线程，在 onDestory()中停止线程。
- 可见的生命周期。从 onStart()开始到 onStop()结束。在这段时间，可以看到 Activity 在屏幕上，尽管有可能不在前台，不能和用户交互。在这两个接口之间，需要保持显示给用户的界面数据和资源等，例如：可以在 onStart()中注册一个 IntentReceiver 来监听数据变化导致界面的变动，当不再需要显示的时候，可以在 onStop()中注销它。onStart()、onStop()都可以被多次调用，因为 Activity 随时可以在可见和隐藏之间转换。
- 前台的生命周期。从 onResume()开始到 onPause()结束。在这段时间里，该 Activity 处于所有 Activity 的最前面，和用户进行交互。Activity 可以经常性地在 Resumed 和 Paused 状态之间切换，例如：当设备准备休眠时，当一个 Activity 处理结果被分发时，当一个新的 Intent 被分发时。所以在这些接口方法中的代码应该属于非常轻量级的。

Activity 的整个生命周期的状态转换和动作都定义在的回调方法中，所有方法都可以被重写，见代码 2.4。

代码 2.4　Activity 的回调方法

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle icicle);
    protected void onStart();
    protected void onRestart();
    protected void onResume();
    protected void onPause();
```

```
    protected void onStop();
    protected void onDestroy();
}
```

下面继续上一小节的例子，在 MainActivity 中重写 Acticity 所有的回调方法，在 ScrollView 中的 TextView 内显示自己在各种状态转换时，调用了哪些不同回调方法。然后可以根据所调用的回调方法，参考图 2.5，确认 Activity 在哪些状态之间进行了转换，何时是何种状态。代码 2.5 是 MainActivity 最终完整的代码。

代码 2.5　MainActivity.java

```
package cn.edu.uibe.c02activitylifecycle;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends Activity {
    private TextView msgtxt;
    private String msgstr;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        msgtxt=(TextView)findViewById(R.id.txt_status);

        msgstr="onCreate() is running…\n";
        msgtxt.setText(msgstr);
    }

    @Override
    protected void onStart() {
        super.onStart();
        msgstr=msgstr+"onStart() is running…\n";
        msgtxt.setText(msgstr);
    }

    @Override
    protected void onRestart() {
        super.onRestart();
        msgstr=msgstr+"onRestart() is running…\n";
```

```
        msgtxt.setText(msgstr);
    }

    @Override
    protected void onResume() {
        super.onResume();
        msgstr=msgstr+"onResume() is running…\n";
        msgtxt.setText(msgstr);
        msgstr="";
    }

    @Override
    protected void onPause() {
        super.onPause();
        msgstr=msgstr+"onPause() is running…\n";
        msgtxt.setText(msgstr);
    }

    @Override
    protected void onStop() {
        super.onStop();
        msgstr=msgstr+"onStop() is running…\n";
        msgtxt.setText(msgstr);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        msgstr=msgstr+"onDestrpy() is running…\n";
        msgtxt.setText(msgstr);
    }

    //处理单击 DIALOG 按钮事件,启动 DialogActivity
    public void startDialog(View v) {
        Intent intent=new Intent(MainActivity.this, DialogActivity.class);
        startActivity(intent);
    }

    //处理单击 DIALOG 按钮事件,启动 ImageActivity
    public void startImage(View v) {
        Intent intent=new Intent(MainActivity.this, ImageActivity.class);
        startActivity(intent);
    }
}
```



2.1.3 任务和回退栈

一个应用程序通常包含多个 Activity。每个 Activity 都可以设计完成特定的用户操作,并且能够启动其他 Activity。例如,一个电子邮件的应用程序可能有一个 Activity,用于展现出新的电子邮件列表,当用户选择了一个电子邮件,就打开一个新的 Activity 以查看该电子邮件的详细内容。

一个 Activity 也可以启动设备上的另一应用程序中的 Activity。例如,如果我们的应用程序想要发送一个电子邮件,可以把邮件地址和内容等信息打包在一个称为 Intent 的组件中,设置启动 Email 应用程序的"创建邮件"Activity,并获取 Intent 中传递的信息。当邮件被发送后,Activity 则重新展现,而用户的感觉是发送邮件的功能好像是应用程序的一部分。虽然上述完成的动作来自不同的应用程序,但是 Android 系统将这些 Activity 放入到相同的任务中,这样就维护了一个完整的用户体验。

所谓任务,就是某些参与用户交互的 Activity 集合,其目的是为完成某项确定的工作。Android 系统通过栈结构来管理任务中的这些 Activity。Activity 按照被打开的顺序排列在栈中。

设备的主(Home)屏幕是大多数任务的起点。当用户触摸应用程序的图标(或者主屏幕上的快捷方式)时,该应用程序的任务就会来到前台。如果该应用的任务不存在(即应用在最近时间段内没有使用过),那么一个新的任务被创建,应用的主 Activity 会作为栈中的根 Activity 打开。

如果用户从当前的 Activity 打开了一个新的 Activity,则新的 Activity 被压入到栈的顶部,并且成为用户的前端界面。而原来的 Activity 仍然在栈中,但是已经变成停止状态,此时系统会保留其用户界面的状态。当用户按下返回按钮时,当前的 Activity 就会从栈的顶部弹出(即当前的 Activity 就会被销毁),而原来的 Activity 就会被重新恢复显示(其界面的状态被系统保存)。在栈中的 Activity 永远不会被重排,只有压入和弹出操作。这种栈的读写方式为"后进先出",我们称其为回退栈。图 2.6 展示了多个 Activity 切换的过程。

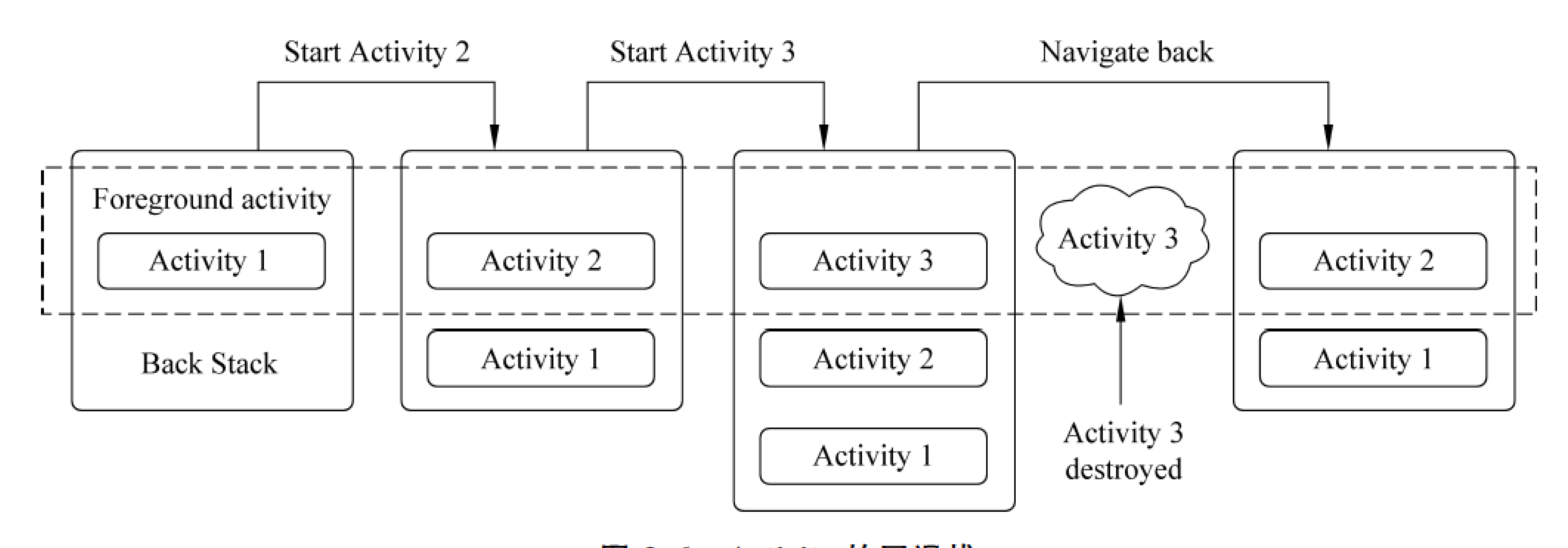
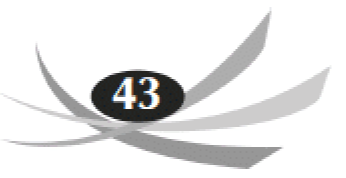


图 2.6 Activity 的回退栈

如果用户不停地按返回键的时候,那么栈中每个 Activity 都会依次弹出,并显示之前



的 Activity，直至用户回到主屏幕（或者当任务启动时的任何一个 Activity）。当所有的 Activity 都从栈中弹出后，这个任务就不再存在。

一个任务就是一个完整的单元，当用户启动一个新的任务时或者使用 Home 按钮回到 Home 屏幕的时候，这个任务就会转变为后台。当任务处于后台时，里面所有的 Activity 都处于停止状态，但是这个任务的回退栈仍然被完整保留。当其他任务变成前台时，当前的任务就变成后台，见图 2.7。

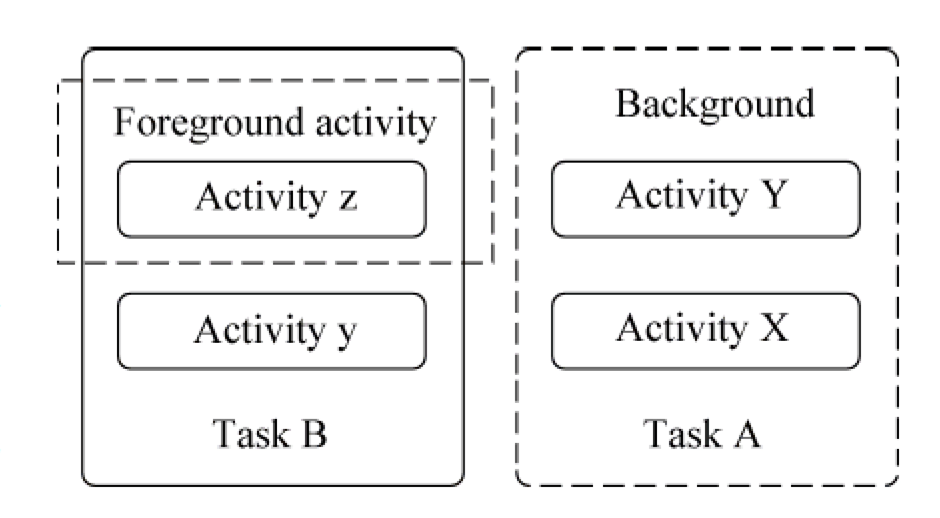


图 2.7　任务的前台与后台

任务可以回到前台，以便用户继续之前的操作。例如，当前任务 A 共有两个 Activity 在栈中，这时用户按下 Home 键切换到主屏幕，然后启动一个新的应用程序。当主屏幕显示时，任务 A 进入后台；而当新的应用程序启动时，系统会为他开启一个新的任务 B，其拥有自己的回退栈。如果用户使用这个应用后，用户使用 Home 键再次切换到主屏幕，并且选中那个启动任务 A 的应用程序。此时，任务 A 进入前台，而其栈中的两个 Activity 仍然被完整保留，并且位于栈顶的 Activity 重新恢复显示。此时，用户仍然可以从主屏幕切换到启动任务 B 的应用程序。这也是 Android 系统中多任务的例子。

虽然 Android 系统可以在后台同时保留多个任务，但是假如用户同时运行着多个后台任务时，系统可能会销毁后台 Activity 用于释放内存，这样的情况就会导致 Activity 状态的丢失。

2.2　理解布局

前一节我们对 Activity 有了初步的了解和理解，也在具体的例子中运用到了 Actiivty 的布局与控件。从继承的概念上来说，Activity 中具体用户图形界面的组件由 Android 定义的 View 类和 ViewGroup 类的子类对象组成，我们把它们称为 View 和 ViewGroup 对象。

View 对象是 Android 平台上用户界面中的基础单元，也可称为控件。Android 系统提供了许多类型的 View，例如 TextView 和 Button 等类，它们都是 View 类的子类。

ViewGroup 对象可以理解为一种容器，类似于 Java 中的 Panel，用于容纳其他的控件对象，并规定这些控件对象按照特定的规则进行排列，即按照某种层次结构排列。Android 系统也提供了许多类型的 ViewGroup，例如 ScrollView、RelativeLayout 和 TabHost 等，它们都是 ViewGroup 的子类。

那么什么是布局呢？View 和 ViewGroup 对象在 Activity 中的排列层次结构，称为用户界面的布局。最常用的是线性布局 LinearLayout、表格布局 TableLayout、相对布局 RelativeLayout、网页布局 WebView 和列表 ListView 等。

在 Android 平台，一个 Activity 的用户界面能够使用层次关系的 View 和 ViewGroup 对象组合来设计布局，例如图 2.8。

在第 1 章我们提到过，Android 系统实现 Activity 的用户界面布局有两种定义方式：一种是使用 XML 文件定义布局，把布局文件置于/res/layout 目录下；另一种是在 Java 应用程序中通过编程的方法来创建 View 和 ViewGroup 对象，在运行时实例化布局元素，或改变其属性。

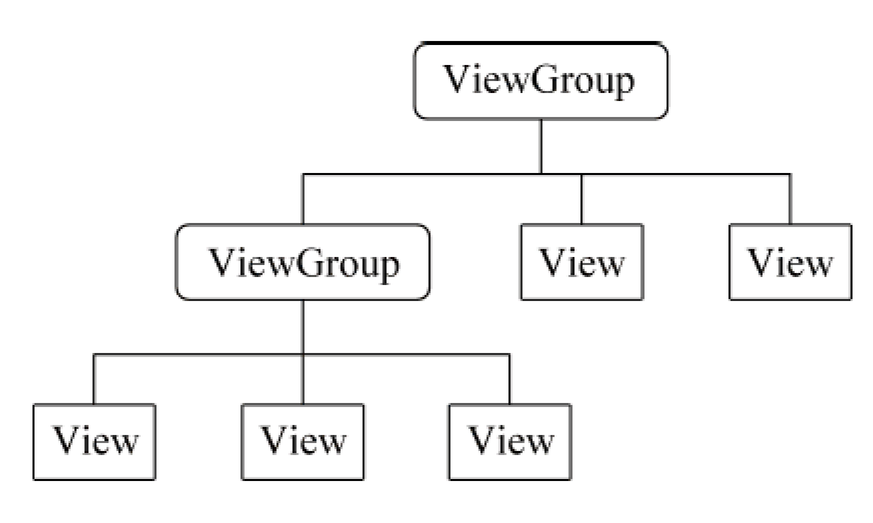


图 2.8　View 和 ViewGroup 对象的布局实例

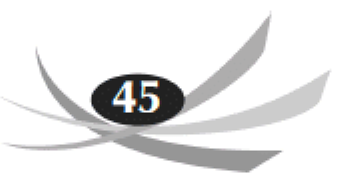
Android 的 XML 布局资源文件主要用于 Activity 用户界面或其他的用户界面组件的布局。使用 XML 布局文件的优势除了在前面提到的外，最重要的是可以将应用程序的界面设计与控制逻辑分离开来，这更有利于用户屏幕不确定的移动应用。如果需要调整界面设计，只需要修改 XML 文件，而无须修改源代码并重新编译。例如，对于不同移动设备或用户、不同的屏幕方向、不同的屏幕尺寸、不同的语言等，我们可以设计不同 XML 布局文件，但是可能并不需要修改任何应用程序代码。此外，对于一个初学者来说，使用 XML 布局更容易定义用户界面的结构，更容易进行调试。

这一节系统地介绍如何在 XML 布局文件中使用 XML 语言，来设计和描述用户图形界面。XML 布局资源文件的具体语法结构见代码 2.6。

代码 2.6　布局资源文件的语法

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@[+][package:]id/resource_name"
    android:layout_height=["dimension" | "fill_parent" | "wrap_content"]
    android:layout_width=["dimension" | "fill_parent" | "wrap_content"]
    [ViewGroup-specific attributes]>
    <View
        android:id="@[+][package:]id/resource_name"
        android:layout_height=["dimension" | "fill_parent" | "wrap_content"]
        android:layout_width=["dimension" | "fill_parent" | "wrap_content"]
        [View-specific attributes]>
        <requestFocus/>
    </View>
    <ViewGroup>
        <View />
    </ViewGroup>
    <include layout="@layout/layout_resource"/>
</ViewGroup>
```

Android 系统的布局 Layout 是 ViewGroup 的子类，根据多样的用户界面要求，Android 系统设计了多种的不同的布局结构，例如 AbsoluteLayout、AdapterView <Textends Adapter >、CoordinatorLayout、DrawerLayout、FragmentBreadCrumbs、



FrameLayout、GridLayout、LinearLayout、LinearLayoutCompat、PagerTitleStrip、RecyclerView、RelativeLayout、SlidingDrawer、SlidingPaneLayout、SwipeRefreshLayout、Toolbar、TvView、ViewPager 等，这些布局进行不同的组合嵌套，则能提供更加丰富的界面布局。

下面以目前最常使用的基本布局线性布局、相对布局和表格布局为例，对布局设计和使用进行详细描述。

2.2.1　线性布局 LinearLayout

线性布局是基础的，使用得比较多的布局类型之一。线性布局的作用就像其名字一样，根据设置的垂直或水平的属性值，将所有的子控件按垂直或水平方向进行组织排列。当布局方向设置为垂直时，布局里面的所有子控件被组织在同一列中；当布局方向设置为水平时，所有子控件被组织在一行中，设置线性布局方向的属性为"android:orientation"，其值可以为 horizontal 或 vertical，分别代表水平或垂直方向，见代码 2.7。

代码 2.7　LinearLayout 语法格式

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal"
android:layout_width="fill_parent" android:layout_height="wrap_content">

<!--add children here-->

</LinearLayout>
```

在这段代码中，还设置了 android:layout_width 和 android:layout_height 属性，分别代表了布局的宽度和高度，这两个属性的值可以为 fill_parent，其代表将视图扩展以填充所在容器（也就是父容器）的全部空间。还可以使用 android:gravity 属性设置布局内组件的对齐方式，其值可以为 top、buttom、left、right、center_vertical 等。

设置边距布局的参数有 layout_marginBottom、layout_marginLeft、layout_marginRight 和 layout_marginTop，分别代表离某元素底边缘、左边缘、右边缘和顶边缘的距离。Android 的 Margin 和 Padding 跟 HTML 的是一样的，见图 2.9。

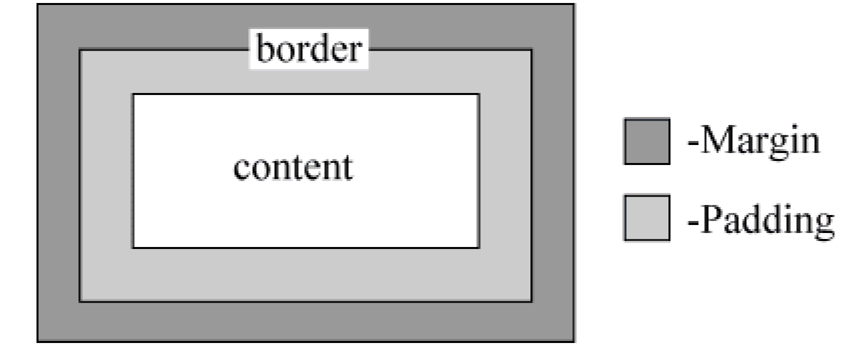


图 2.9　布局划分的参数定义

通俗地理解 Padding 为内边框，Margin 为外边框，代码 2.8 显示了如何设置一个线性布局的边框。

代码 2.8　设置 LinearLayout 边框

```
android:layout_marginBottom="25dip"
android:layout_marginLeft="10dip"
android:layout_marginTop="10dip"
```

```
android:layout_marginRight="10dip"
android:paddingLeft="1dip"
android:paddingTop="1dip"
android:paddingRight="1dip"
android:paddingBottom="1dip"
```

如果左、右、上、下都是相同的设置，则可以按照如下代码直接设置。

```
android:layout_marginBottom="25dip"
android:layout_margin="10dip"
android:padding="5dip"
```

LinearLayout 所定义的界面上，所有的子元素都被堆放在其他元素之后，因此一个垂直列表的每一行只会有一个元素，而一个水平列表将会只有一个行高。

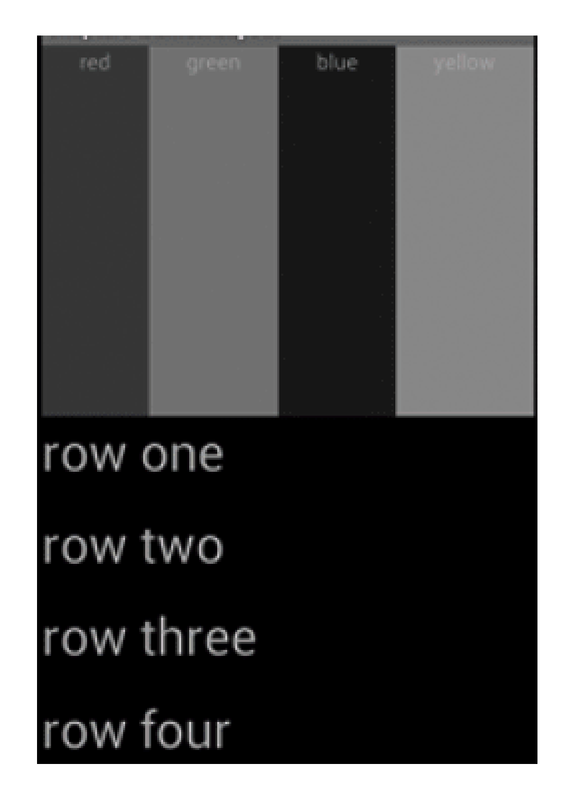


图 2.10　线性布局实例效果

LinearLayout 的可选属性 layout_weight，能够指定每个子控件在父级线性布局中的相对重要程度。LinearLayout 还支持为单独的子元素指定 weight，这样避免了在一个大屏幕中一串小对象挤成一堆的情况，而是允许它们放大填充空白。子元素指定一个 weight 值，剩余的空间就会按这些子元素指定的 weight 比例分配给这些子元素。默认的 weight 值为 0。例如，如果有三个文本框，其中两个指定了 weight 值为 1，那么，这两个文本框将等比例地放大，并填满剩余的空间，而第三个文本框不会放大，见图 2.10。

要实现这个界面，需要下面几个步骤：

(1) 在 Android 项目的 src 目录下，创建显示界面的 LinearLayoutActivity 类，见代码 2.9。

(2) 创建布局文件 linear_layout.xml，存放在/res/layout 目录下，见代码 2.10。

(3) 修改 AndroidManifest.xml 文件，在其中添加 LinearLayoutActivity 的声明，见代码 2.11。

代码 2.9　LinearLayoutActivity.java

```
public class LinearLayoutActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        //TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.linear_layout);
    }
}
```

代码 2.9 中的 setContentView(R.layout.linear_layout)表示把布局文件 linear_layout.xml 中定义的控件和排列显示在 LinearLayoutActivity 定义的 Activity 中。

代码 2.10　linear_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical">

    <LinearLayout
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:layout_weight="1"
     android:orientation="horizontal">

        <TextView
         android:layout_width="wrap_content"
         android:layout_height="fill_parent"
         android:layout_weight="1"
         android:background="#aa0000"
         android:gravity="center_horizontal"
         android:text="red" />

        <TextView
         android:layout_width="wrap_content"
         android:layout_height="fill_parent"
         android:layout_weight="1"
         android:background="#00aa00"
         android:gravity="center_horizontal"
         android:text="green" />

        <TextView
         android:layout_width="wrap_content"
         android:layout_height="fill_parent"
         android:layout_weight="1"
         android:background="#0000aa"
         android:gravity="center_horizontal"
         android:text="blue" />

        <TextView
         android:layout_width="wrap_content"
         android:layout_height="fill_parent"
```

```
        android:layout_weight="1"
        android:background="#aaaa00"
        android:gravity="center_horizontal"
        android:text="yellow" />
    </LinearLayout>

    <LinearLayout
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:layout_weight="1"
     android:orientation="vertical">

       <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="row one"
        android:textSize="15pt" />

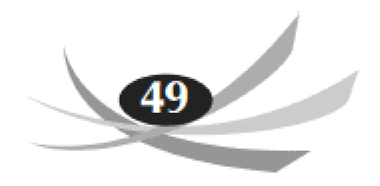
       <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="row two"
        android:textSize="15pt" />

       <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="row three"
        android:textSize="15pt" />

       <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="row four"
        android:textSize="15pt" />
    </LinearLayout>

</LinearLayout>
```

代码 2.10 中定义了三个线性布局。外层的线性布局，通过“android:orientation＝



"vertical"”定义布局内的空间按垂直方向排列。这个外层布局中有两个控件，分别是设置成水平方向和设置成垂直方向的两个线性布局，第一个线性布局中是 4 个设置成不同颜色的 TextView 控件，按水平方向排列；第二个线性布局中是 4 个设置成"row one"等不同文本的 TextView，按垂直方向排列。

完成布局文件的定义后，就可以在 AndroidManifest. xml 文件中添加显示这个界面的 LinearLayoutActivity。

代码 2.11　LinearLayoutActivity 的声明

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="mc.sample"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk android:minSdkVersion="14" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name">
        <activity
            android:name=".HelloWorldActivity"
            …
        </activity>
        <activity
            android:name=".LinearLayoutActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

完成 Activity 的声明之后，在应用程序中就可以运行定义好的 LinearLayoutActivity 了。

2.2.2　相对布局 RelativeLayout

RelativeLayout 允许布局中的控件根据其他控件或布局本身的相对位置来指定如何排列。因此，可以使用以右对齐、上下或置于屏幕中央等形式来排列两个元素。布局中的控件是按顺序排列的，如果第一个元素在屏幕的中央，那么相对于这个元素的其他元素将以屏幕中央的相对位置来排列。如果使用 XML 布局文件来定义这种布局，之前被关联

的元素必须定义。

RelativeLayout 布局的相关属性见表 2.1。

表 2.1 RelativeLayout 布局的相关属性

属　　性	含　　义
android:layout_above	将该控件的底部置于给定 ID 控件之上
android:layout_below	将该控件的底部置于给定 ID 控件之下
android:layout_toLeftOf	将该控件的右边缘与给定 ID 控件左边缘对齐
android:layout_toRightOf	将该控件的左边缘与给定 ID 控件右边缘对齐
android:layout_alignBaseline	将该控件的基准线与给定 ID 基准线对齐
android:layout_alignTop	将该控件的顶部边缘与给定 ID 顶部边缘对齐
android:layout_alignBottom	将该控件的底部边缘与给定 ID 底部边缘对齐
android:layout_alignLeft	将该控件的左边缘与给定 ID 左边缘对齐
android:layout_alignRight	将该控件的右边缘与给定 ID 右边缘对齐
android:layout_alignParentTop	如果为 true,将该控件的顶部与其父控件的顶部对齐
android:layout_alignParentBottom	如果为 true,将该控件的底部与其父控件的底部对齐
android:layout_alignParentLeft	如果为 true,将该控件的左部与其父控件的左部对齐
android:layout_alignParentRight	如果为 true,将该控件的右部与其父控件的右部对齐
android:layout_centerHorizontal	如果为 true,将该控件的中央置于水平居中
android:layout_centerVertical	如果为 true,将该控件的中央置于垂直居中
android:layout_centerInParent	如果为 true,将该控件的中央置于父控件的中央
android:layout_marginTop	上偏移的值
android:layout_marginBottom	下偏移的值
android:layout_marginLeft	左偏移的值
android:layout_marginRight	右偏移的值

图 2.11 是采用 RelativeLayout 布局显示的效果。

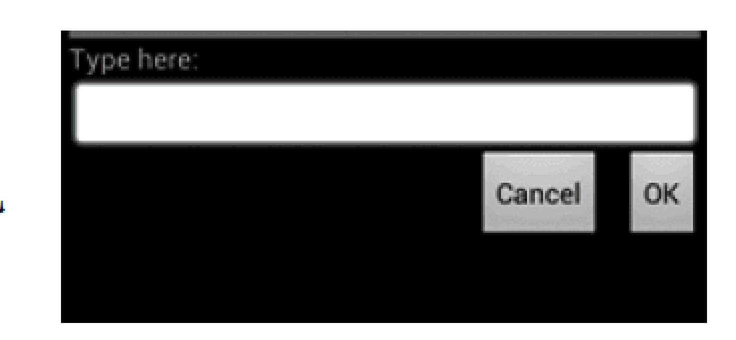


图 2.11 相对布局实例效果

要实现这个界面,需要下面几个步骤:

(1) 创建显示界面的 RelativeLayoutActivity 类,见代码 2.12。

(2) 创建布局文件 relative_layout.xml,存放在/res/layout 目录下,见代码 2.13。

(3) 修改 AndroidManifest.xml 文件,在其中添加 RelativeLayoutActivity 的声明。

代码 2.12　RelativeLayoutActivity.java

```
import cn.edu.uibe.mc..sample.R;
import android.app.Activity;
import android.os.Bundle;

public class RelativeLayoutActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        //TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.relative_layout);
    }
}
```

代码 1.13　relative_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Type here:" />

    <EditText
        android:id="@+id/entry"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/label"
        android:background="@android:drawable/editbox_background" />

    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_below="@id/entry"
        android:layout_marginLeft="10dip"
```

```
        android:text="OK" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@id/ok"
        android:layout_toLeftOf="@id/ok"
        android:text="Cancel" />

</RelativeLayout>
```

2.2.3 表格布局 TableLayout

TableLayout 把用户界面按表格形式划为行和列，然后把控件分配到指定的行或列中。一个 TableLayout 由许多 TableRow 组成，每个 TableRow 定义一行 row。TableLayout 容器不会显示行、列或单元格 cell 的边框线。每行可有 0 个或多个 cell；每个 cell 能容纳一个 View 对象。表格允许 cell 为空，但 cell 不能跨列。

```
android:collapseColumns
```

以第 0 行为序，隐藏指定的列：设置 android:collapseColumns=0,2 意思是把第 0 列和第 2 列隐藏。

```
android:shrinkColumns
```

以第 0 行为序，自动延伸指定的列填充可用部分：当 LayoutRow 里面的控件还没有布满布局时，shrinkColumns 不起作用，设置 shrinkColumns=0,1,2，布局完全没有改变，因为 LayoutRow 里面还剩足够的空间。当 LayoutRow 布满控件时，设置 shrinkColumns=2，则控件自动向垂直方向填充空间。

```
android:stretchColumns
```

以第 0 行为序，尽量把指定的列填充空白部分：设置 stretchColumns=1，则结果见图 2.12，第 1 列被尽量填充（Button02 与 TextView02 同时向右填充，直到 TextView03 被压挤到最后边）。



图 2.12 表格布局实例效果

要实现图 2.12 所示的界面，需要下面几个步骤：

(1) 创建显示界面的 TableLayoutActivity 类；

(2) 创建布局文件 table_layout.xml，存放在 /res/layout 目录下，见代码 2.14；

(3) 修改 AndroidManifest.xml 文件，在其中添加 TableLayoutActivity 的声明。

代码 2.14 table_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">

    <TableRow>
        <TextView
            android:layout_column="1"
            android:padding="3dip"
            android:text="Open…" />
        <TextView
            android:gravity="right"
            android:padding="3dip"
            android:text="Ctrl-O" />
    </TableRow>

    <TableRow>
        <TextView
            android:layout_column="1"
            android:padding="3dip"
            android:text="Save…" />
        <TextView
            android:gravity="right"
            android:padding="3dip"
            android:text="Ctrl-S" />
    </TableRow>

    <TableRow>
        <TextView
            android:layout_column="1"
            android:padding="3dip"
            android:text="Save As…" />
        <TextView
            android:gravity="right"
            android:padding="3dip"
            android:text="Ctrl-Shift-S" />
    </TableRow>

    <View
        android:layout_height="2dip"
        android:background="#FF909090" />
```

```
    <TableRow>
        <TextView
            android:padding="3dip"
            android:text="X" />
        <TextView
            android:padding="3dip"
            android:text="Import…" />
    </TableRow>

    <TableRow>

        <TextView
            android:padding="3dip"
            android:text="X" />
        <TextView
            android:padding="3dip"
            android:text="Export…" />
        <TextView
            android:gravity="right"
            android:padding="3dip"
            android:text="Ctrl-E" />
    </TableRow>

    <View
        android:layout_height="2dip"
        android:background="#FF909090" />

    <TableRow>
        <TextView
            android:layout_column="1"
            android:padding="3dip"
            android:text="Quit" />
    </TableRow>

</TableLayout>
```

2.3 使用布局

Android 的用户界面布局是在 XML 文件中静态记载，在 Android 的 Java 程序中动态加载的。当编译 Android 应用程序时，每一个 XML 布局文件被编译成 View 视图资源，应用程序代码在 Activity. onCreate() 回调中实现布局资源的加载，通过调用 setContentView()传递给它的形式引用到布局资源 R. layout. layout_file_name。

例如，如果 XML 布局文件保存于 main_layout.xml 中，实现 Activity 加载的代码如代码 2.15。

代码 2.15　Activity 加载布局文件资源

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

1. 重用布局

Android 用户界面的布局是可以重用的。重用布局的功能非常强大，因为它允许创建可重复使用的复杂的布局。在应用程序中，用户界面布局中相同或类似的任何元素都可以被提取出来，定义成一个独立的布局文件，单独管理，然后在需要的时候嵌入到另一个布局中。例如，一个"是/否"的按钮面板，或自定义的进度条说明文字等，单独定义后可以嵌入到任何其他布局中。因此，程序员可以根据需求灵活地设计自定义的视图，定义自己特殊的布局。

如果要有效地重复使用完整的布局，可以在当前布局使用的'<include/>'和'<merge/>'的标签嵌入到另一个布局。

下面通过一个例子来具体说明如何使用'<include/>'重用布局。

首先创建一个布局文件 titlebar.xml，其中定义了标题栏和 logo，将其作为重用的布局，见代码 2.16。

代码 2.16　titlebar.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/titlebar_bg">

    <ImageView android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:src="@drawable/gafricalogo" />
</FrameLayout>
```

创建另一个布局文件 reuse_titlebar.xml，使用'<include/>'的标签把 titlebar.xml 定义的布局嵌入到这个布局中，见代码 2.17 中带下画线的语句。

代码 2.17　reuse_titlebar.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/app_bg"
```

```
    android:gravity="center_horizontal">

    <include layout="@layout/titlebar"/>

    <TextView android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:text="@string/hello"
              android:padding="10dp" />
    ...
</LinearLayout>
```

<merge>标签在优化 UI 结构时起到很重要的作用。目的是通过删减多余或者额外的层级，从而优化整个 Android Layout 的结构。

<merge>的另外一个用法，就是使用<merge>替代 Layout 标签作为重用布局文件的根节点时，当另一个布局文件使用 Include 或者 ViewStub 标签从外部导入其 XML 结构时，可以很好地将它所包含的子集融合到父级结构中，而不会出现冗余的布局节点。

例如，如果在代码 2.16 中定义的 Layout 是 LinearLayout，则布局的规则与重用它的代码 2.17 相同，使用 include 嵌入布局中的组件，与父节点的其他组件都按照同样的排列规则显示。但是，从整个 Android Layout 的结构来看，就多了一个 LinearLayout 冗余节点。在代码 2.18 中使用<merge>替代根节点，嵌入其他布局文件后就可以直接采用父节点的布局，与父节点的其他组件在同一级结构中。

代码 2.18　merge_layout.xml

```
<merge xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/titlebar_bg">

    <ImageView android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:src="@drawable/gafricalogo" />
</merge>
```

2. 获取控件

在布局文件中定义了界面的布局之后，如果要在应用程序中对控件进行操作，则必须根据布局文件中的定义获取控件对应的对象。例如单击图形界面的一个按钮后，需要对这个按钮的事件进行处理，执行单击按钮后的相应代码。但这个代码写在哪里呢？应用程序中没有这个按钮的定义，也没有创建这个按钮对象。因此，应用程序必须根据布局文件中定义的其 ID 属性，从 Android 系统中获取这个按钮对象，编写其对应的事件处理代码，界面才能够做出正确的响应。

每个 View 和 ViewGroup 对象都有很多各自的属性，有些属性属于特定的 View 对

象，有些属性是所有View对象共同有的。所有对象共同有的属性都是从根View类继承来的，ID属性也是这样。

在XML布局文件的树形结构中，View对象的ID属性是这个控件的唯一标识。在XML布局文件中，这个ID通常表现为一个字符串，当编译应用程序时，这个ID会被引用为一个整数。这是一个公共属性，我们会经常用他。

下面是在XML布局文件中，定义一个控件ID属性的语法：

```
android:id="@ +id/my_button"
```

XML解析器解析@后面的字符串，my_button就是指定的ID字符串，id是指其在R.java文件中的分类；+表示这是一个新的资源名称，必须建立并加入到项目的R.java文件中。

获取控件包括两个步骤：

(1) 在布局文件中定义View或部件，并赋予唯一的ID。

```
<Button android:id="@+id/my_button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/my_button_text"/>
```

(2) 在Android应用程序中，使用findViewById()引用布局文件中的部件，创建一个部件对象。

```
Button myButton=(Button)findViewById(R.id.my_button);
```

2.4　样式和主题

样式是用于指定View或Windows的外观和格式的一系列属性的集合。样式可以指定控件或布局的高度、填充、字体颜色、字体大小、背景颜色等等属性。Android中的样式与网页设计中的层叠样式表(CSS)有着相似的原理，就是允许我们将设计从内容中分离出来。例如，使用一个样式，我们可以将下面这个布局：

```
<TextView
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:textColor="#00FF00"
       android:typeface="monospace"
       android:text="@string/hello" />
```

变成这样：

```
<TextView
    style="@style/CodeFont"
    android:text="@string/hello" />
```

这样，将所有与样式相关的属性从 XML 布局中移出，放到一个名为 CodeFont 的样式定义中，通过样式属性应用。

主题是一个应用于整个 Activity 或应用中，而不是某一个单独的 View。当一个样式被作为主题来应用时，则这个样式对 Activity 或应用中的每个 View 都有效。例如，我们能把 CodeFont 样式作为主题应用于一个 Activity，那么这个 Activity 中所有文本都将是绿色等宽字体。

2.4.1　定义样式

如果要创建一套样式，需要在项目的 res/values/ 目录下创建一个 XML 文件，来定义样式。定义样式的 XML 文件名称由程序员任意指定，但必须使用. xml 作为扩展名，保存在 res/values/ 文件夹中，而且文件中的根节点必须是<resources>。

resources 节点下由 style 子元素定义样式的具体配置，其 name 属性所创建样式的唯一标识。style 元素下可以有多个<item>子元素，具体来定义 View 各属性的配置。<item>元素包含一个 name 属性 和一个对应值，说明这一项设定哪个 View 的属性的样式。<item> 元素本身的值可以是一个关键字符串、十六进制颜色、另一个资源类型的引用或其他值，它是属性的样式值。代码 2. 19 是 CodeFont 的样式定义，可以看出样式定义 XML 文件的结构和语法。

代码 2. 19　style_sample. xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

每个<resources> 元素的子节点在编译时都被转换为一个应用程序资源对象，可通过<style> 元素的 name 属性的值来引用。例如前面代码中的样式，通过 style="@style/CodeFont"语句来引用。

在<style> 元素中的 parent 属性是可选的，让我们能够从指定的 style 中继承所有属性。通过这种途径从一个现有的 style 中继承属性后，可以根据需求改变或添加的属性，从而创建新的样式。例如，下面的样式定义是从 Android 平台默认文本外观样式继承，修改了一下文本的颜色。

```
<style name="GreenText" parent="@android:style/TextAppearance">
    <item name="android:textColor">#00FF00</item>
</style>
```

如果要继承的是自定义的样式，就不必使用 parent 属性，而使用“.”把原有的样式和新样式名连接起来。例如，下面的代码创建了一个新样式，它继承前面定义的 CodeFont，但把颜色改为红色。

```
<style name="CodeFont.Red">
    <item name="android:textColor">#FF0000</item>
</style>
```

这里没有使用 parent 属性，name 属性以 CodeFont 起始，使用“.”连接了后面的新样式名称。这个新样式可以通过@style/CodeFont.Red 来引用。样式的继承可以有多重。例如下面的代码，从 CodeFont 和 CodeFont.Red style 中同时继承，然后添加 android:textSize 属性。

```
<style name="CodeFont.Red.Big">
    <item name="android:textSize">30sp</item>
</style>
```

这种技巧仅适用于将自定义的资源链接起来，不能用这种方式继承 Android 内置的 style。要引用一个 Android 的内置 style，必须使用 parent 属性。

2.4.2　使用样式

定义一个样式之后，如果对一个 View 应用了这个样式，而这个 View 并不支持此样式中设定的某些属性，那么此 View 将应用那些它支持的属性，并简单忽略那些不支持的。

在 Activity 或应用程序中有两种方式来使用 style：

(1) 对一个独立的 View，在布局文件 XML 中将 style 属性添加到的此 View 元素中；

(2) 对一个 Activity 或应用，在 AndroidManifest.xml 文件中将 android:theme 属性添加到的<activity> 或<application> 元素中。

如果将一个 style 应用到布局中一个单独的 View 上时，此 style 定义的属性会仅应用于那个 View。如果一个 style 应用到一个 ViewGroup 上，其子 View 元素并不会继承应用此 style 属性，只有直接设置其子元素此 style，才会起作用。但是，通过第二种方式，将 style 作为 theme 来应用的方式，将会把这个 style 应用到此 Activity 或<application>的所有 View 元素上。

下面是在 XML 布局中为 View 设置 style 的简单语法：

```
<TextView
    style="@style/CodeFont"
    android:text="@string/hello" />
```

如果需要对应用程序中所有 Activity 设置一个 theme，则打开 AndroidManifest.xml 文件并编辑<application> 标签，使之包含 android:theme 属性和 style 名称，具体设置代码如下：

```
<application android:theme="@style/CustomTheme">
```

如果希望 theme 仅应用到应用程序中的某个 Activity 中，那么就将 android:theme 属性添加到<activity> 标签中。

2.5 理解资源

Android 应用程序不仅包括逻辑代码，还包括资源文件，如字符、图片、布局和语言支持等。Android 系统对于资源的管理使用了一种将资源外部化的模式。这种方式，使得应用程序可以在代码编译时，只是使用资源的引用，在代码编译后修改资源包含的内容也不会影响程序的逻辑。从而保持程序逻辑和资源的各自独立。对于外部资源，可以通过提供替代资源的方式，支持不同的语言或屏幕大小。随着越来越多不同配置的 Android 设备的出现，这种模式对于 Android 程序运行于复杂多变的环境尤其重要。

Android 的资源以文件形式，在项目的 res/目录下进行统一管理。为了提供具有不同配置的兼容性，必须在项目的 res/目录中组织资源，在其不同子目录中存放不同的资源类型和配置。对于任何类型的资源，都可以指定默认情况下使用的资源和多个替代资源。

使用默认资源的条件是指可以支持任何配置的 Android 设备或当前的配置没有替代资源匹配，图 2.13 中的界面只设计有一种布局；而替代的资源是为特定配置设计的，图 2.14 中的界面为横向的屏幕设置了替代布局。通过资源文件目录名，Android 系统会自动应用相应的资源文件，匹配设备当前的配置。

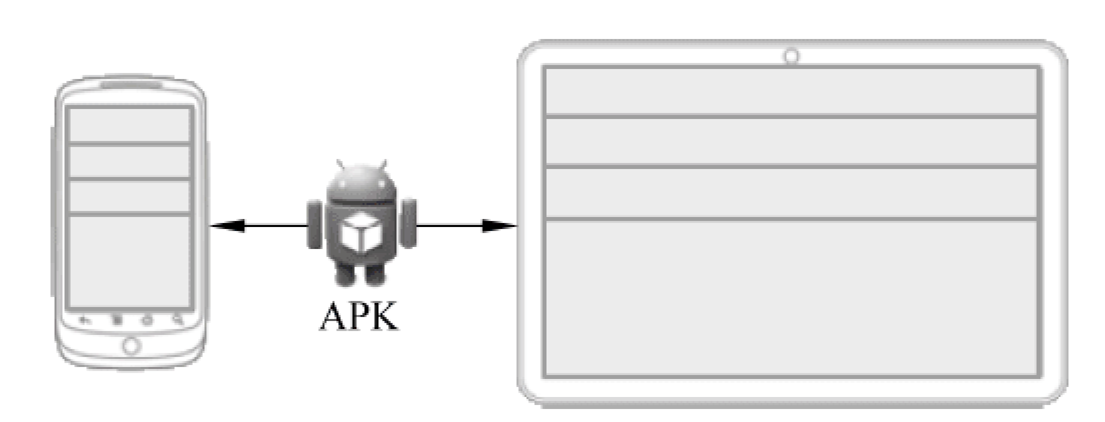


图 2.13　布局设计之一

对于 res/目录下的资源，应用程序可以通过引用资源 ID 号来调用。具体的资源 ID 号能够从 R.java 中查到。对于每一种资源类型都有一个 R 的子类对应着（例如 R.drawable 中包含着所有 drawable 资源），并且对每个特定类型的所有资源都有一个静态的整型数值一一对应（例如 R.drawable.icon）。这个整型数值就是这个特定资源的 ID

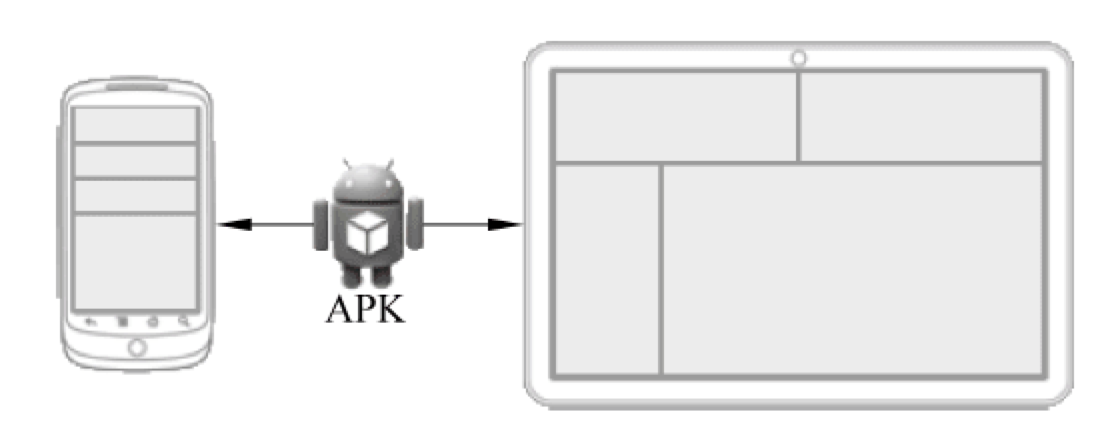


图 2.14　布局设计之二

号，通过它能获取对应的资源。

一个资源的 ID 号一般的组成如下：

- 资源类型。每种资源都会被分组到一种特定的资源类型，例如 string，drawable 和 layout 等，还有更多的资源类型，如 raw，color 等。
- 资源名。资源名同时也是文件名，不包括拓展名；或者是 XML 中 android：name 属性的值，这种情况适用于这个资源是一个简单的值（例如一个字符串）。

2.5.1　提供资源

res/的子目录中包含了所有资源，资源目录名是很重要的，表 2.2 列出了资源目录和具体资源类型的对照。

表 2.2　资源目录和资源类型

目　录	资 源 类 型
animator/	存放定义属性动画的 XML 文件
anim/	存放定义补建动画的 XML 文件
color/	存放定义颜色值的 XML 文件
drawable/	存放位图文件（.png、.jpg、.gif、.9.png），或者是被编译成可描画资源类型的 XML 文件
layout/	存放定义用户界面布局的 XML 文件
menu/	存放定义应用程序菜单的 XML 文件，如选项菜单、上下文菜单或子菜单
raw/	存放任意原生格式的文件
values/	存放包含简单值的 XML 文件，如字符串、整数以及颜色等
xml/	放在这个目录下的任意 XML 文件，都可在运行时通过调用 Resources.getXML() 方法来读取

值得注意的是，不能把资源文件直接保存在 res/目录中，这样会导致编译错误。

保存在表 2.2 中所定义子目录中的资源是默认资源。也就是说，这些资源定义了 Android 应用程序用户界面的默认设计和内容。

但是，对于同一个应用程序来说，可以根据 Android 设备的不同特性和设置预先定义不同类型的资源，以方便应用程序的用户界面切合运行时的硬件设备。例如，可以针对竖屏和横屏设定不同的布局资源文件，以满足屏幕切换的需要。或者也可以针对不同的语

言,提供不同的字符串资源,使得在用户界面上显示与设备语言相匹配的文字。要给不同的设备配置提供这些不同的资源,除了默认的资源以外,还要提供可选的替代资源。

2.5.2 访问资源

Android 应用程序引用某个资源时,有两种方法。

(1) 在 Java 应用程序代码中直接调用,通过调用 Resources 类中的方法来获取某一特定的资源,通过 getResources()方法得到 Resources 类的一个实例。在应用程序代码中引用资源的语法如下:

```
[<package_name>.]R.<resource_type>.<resource_name>
```

其中,＜package_name＞指资源所在的包名,如果资源文件在项目本身的包内时,该字段不需要填写。＜resource_type＞指 R 类下对应一种特定资源类型的子类,如 R. String。＜resource_name＞可以是不包含文件扩展名的资源文件名或者 XML 元素中 android:name 属性的值。

例如,Java 应用程序中的语句"R. drawable. my_background_image"调用了资源类型为 drawable,资源名为 my_background_image 的 Android 资源。实际上就是引用了 Android 定义好的图片资源。

```
//使用 drawable 类型的图片资源给当前屏幕加载背景
    getWindow().setBackgroundDrawableResource
(R.drawable.my_background_image);
//使用 Layout 类型的布局资源作为当前屏幕的布局
    setContentView(R.layout.main_screen);
```

(2) 在 XML 中调用,通过特殊的 XML 语法引用 R. class 文件中的相关资源 ID。在 XML 资源文件中引用资源的语法如下:

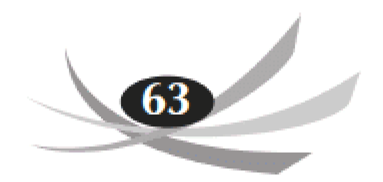
```
@[<package_name>:]<resource_type>/<resource_name>
```

语句中各标记的含义同上。

例如,XML 布局文件中的语句"@color/opaque_red"和"@string/hello"调用了资源类型分别为 Color 和 String,资源名分别为 opaque_red 和 hello 的 Android 资源。

```
//使用 Color 资源类型的 opaque_red 的颜色资源作为文本颜色
android:textColor="@color/opaque_red"
//使用 String 资源类型的 hello 的字符串作为文本显示的内容
android:text="@string/hello" />
```

上面语句中的资源定义都在本项目中,这种情况不需要说明包。如果要引用 Android 系统定义的资源,则需要包含包名。例如:




```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@android:color/secondary_text_dark"
    android:text="@string/hello"/>
```

应该在任何时候都使用字符串资源，以便应用程序能够针对其他语言进行本地化。

我们可以在任何需要使用自己提供的资源的地方，通过这两种语法来调用。

在 Android 系统中，不仅仅资源本身可以被引用，在定义样式时，也可以引用样式属性。引用样式属性的语法与普通的资源格式几乎是等同的，但是使用问号“?”取代符号“@”，资源类型部分是可选的，语法格式如下：

```
?[<package_name>:][<resource_type>/]<resource_name>
```

下面的例子是引用 Android 设定的一个样式属性 textColorSecondary 来设置布局中 TextView 的文本颜色，使得其匹配系统主题的“主”文本的颜色。这里不需要说明样式属性的资源类型。

```
<EditText id="text"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="?android:textColorSecondary"
android:text="@string/hello_world" />
```

Android 中包含有很多标准的资源，例如 styles（样式）、themes（主题）、layouts（布局）等。要调用这些资源，需要通过 Android 包名来限定这些资源。

2.6　多屏幕适应

移动终端的类型繁多，屏幕的尺寸、长宽大小比例也多种多样。Android 应用程序要运行在不确定的屏幕上，Android 系统需要处理适配不同显示屏幕的工作。因此 Android 系统对不同的屏幕尺寸和密度提供 API，应用程序能够设计提供不同的屏幕尺寸和密度的用户界面。应用程序运行时，系统通过适当的管理方式，针对当前的屏幕配置引用与对应的资源进行适配，调整布局和位图。

如果没有替代的布局资源，系统则采用屏幕默认设计，并根据屏幕的大小进行缩放和调整。但通过提供多种屏幕布局的方式，可以最大程度优化用户体验。

Android 系统是怎样支持多种屏幕的呢？在应用程序用户界面设计中，可以从以下几个方面来设定对多种屏幕的支持。

1. 明确声明应用程序支持的所有屏幕尺寸

在 AndroidManifest.xml 文件中，使用＜supports-screens＞元素说明应用程序能够

支持的屏幕尺寸。通过声明应用程序支持的屏幕尺寸，可以保证只有那些屏幕尺寸被应用程序支持的设备才可以下载该应用程序。

2. 为不同的屏幕尺寸提供不同的布局

默认情况下，Android 可以重新调整应用程序的布局，以适应当前的设备屏幕。但有时候可能调整后的屏幕布局不太美观，这就需要应用程序针对特殊的屏幕（过大或过小）设计不同的布局。在大多数情况下，这是可行的。使用配置限定符能够提供与尺寸相关的资源，这些限定符包括 small、normal、large 和 xlarge。如果应用程序运行在 Android 3.2（API Level 13）以上，则需要使用 sw<N>dp 配置限定符定义布局资源所需的最小可用宽度。例如 600dp 屏幕以上的布局，放置在 layout-sw600dp/目录下。下面代码就是一个设定布局的简单例子：

```
res/layout/my_layout.xml    //layout for normal screen size("default")
res/layout-small/my_layout.xml         //layout for small screen size
res/layout-large/my_layout.xml         //layout for large screen size
res/layout-xlarge/my_layout.xml        //layout for extra large screen size
res/layout-xlarge-land/my_layout.xml
                          //layout for extra large in landscape orientation
```

3. 为不同的屏幕尺寸提供不同的位图

默认情况下，Android 系统会在应用程序运行时，根据运行设备的屏幕缩放位图（.png、.jpg 和.gif 文件）和 Nine-Patch（.9.png 文件），使它们显示出合适的物理尺寸。但这种缩放有时会失真。最好的方法是为不同的屏幕密度提供不同分辨率的位图。

Android 系统使用限定符 ldpi（low）、mdpi（medium）、hdpi（high）和 xhdpi（extra high），来提供密度相关的资源。见如下代码，为高密度屏幕提供的位图放在 drawable-hdpi/目录下。

```
res/drawable-mdpi/my_icon.png        //bitmap for medium density
res/drawable-hdpi/my_icon.png        //bitmap for high density
res/drawable-xhdpi/my_icon.png       //bitmap for extra high density
```

当 Android 应用程序运行时，系统怎样使用适当的可选资源呢？系统会根据当前屏幕的大小和密度配置，查找应用程序提供的布局和密度相关资源，确定最佳匹配的资源目录下的资源。例如，应用程序在一个高密度的大屏幕上显示一个 drawable 资源，系统会寻找最接近此大屏幕的布局、高密度 drawable 目录的资源来使用。

如果没有匹配的资源是可用的，系统将使用默认的资源并且对其进行缩放来适应当前的屏幕尺寸和密度。默认的资源是那些没有配置限定符的资源。

2.7 本章小结

本章主要介绍了 Activity 的概念和生命周期，以及如何使用 Activity 类创建用户界面。Activity 是 Android 的四大基本组件之一，通过 Activity，用户可以与移动终端进行

交互。Activity 的生命周期中有 Active/Running、Paused、Stopped 和 Killed 四种状态。

本章还着重介绍了布局概念和分类。Activity 中的具体图形控件由 Android 定义的 View 类和 ViewGroup 类的子类对象组成，这些对象在 Activity 中的排列结构，称为用户界面的布局。Android 有五种基本的布局对象：FrameLayout（框架布局）、LinearLayout（线性布局）、AbsoluteLayout（绝对布局）、RelativeLayout（相对布局）和 TableLayout（表格布局）。Android 的用户界面布局是在 XML 文件中静态记载，在 Android 的 Java 程序中动态加载的。在这一章中，针对每一种基础布局，使用具体的代码实现说明了如何在用户界面中使用这些布局。

在本章的偏后部分介绍了如何使用 Android 项目中的样式和资源的概念。

第 3 章 Fragment 和图形控件

3.1 理解 Fragment

Android 运行在各种各样的设备中，有各种尺寸的手机、各种标准的平板电脑、各种类型的控制屏幕甚至电视，而且在使用移动设备时，用户还会经常转换屏幕的纵横角度。虽然可以通过提供不同尺寸的位图、声明多种屏幕尺寸来解决分辨率不同的问题，但如果要 App 的界面友好美观，还需要做大量工作来为同一界面设计多个布局，实现多屏幕适应。

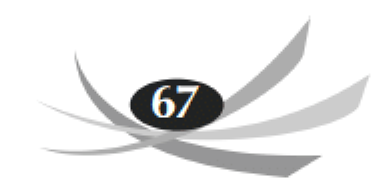
在开发过程中，一般都是先基于手机开发一套 App，然后复制一份，修改布局以适应超级大屏幕。难道无法做到一种 UI 的布局可以同时适应手机和平板吗？在手机的 UI 布局设计过程中，主要是定义其中 View 和 ViewGroup 对象的层级结构。如果把这些结构模块化，就可以直接在屏幕转换角度或在平板电脑等大屏幕的布局上使用了。Fragment 出现的初衷就是为了解决这样的问题。

3.1.1 Fragment 的概念

Fragment 是 Activity 中用户界面的一部分，Fragment 是一种灵活的、可重用 UI 组件。可以把一个 Fragment 看成是 Activity 的一个布局模块，概念上可以理解成 Java 语言中 Frame 中的 Panel，可以容纳具有某种层次结构的 View 和 ViewGroup 对象。

Android 系统从 Android 3.0(API level 11)开始推出 Fragment，主要目的是支持多屏幕更加动态和灵活的 UI 设计。用户界面设计时，把一个 Activity 切分成多个 Fragment，那么在把 App 从手机屏幕迁移到大屏幕时，布局设计就不再需要改变和管理 Fragment 内部的 View 层次结构，而可以重点考虑 Activity 的显示外观了。

例如，一个应用在手机中使用 Activity A 显示文章列表，Activity B 显示列表中对应的文章内容。当用户单击 Activity A 中文章列表的一项时，Activity A 启动 Activity B，在 Activity B 中显示选中的文章内容，见图 3.1 右图。如果每一个 Activity 都使用一个 Fragment 来容纳界面上的 View 对象，在设计这个应用在平板电脑的布局时，就可以不必考虑具体每一个图形控件的排列位置和关系，只需要考虑屏幕的容纳空间，适合排列哪些 Fragment，考虑用户操作时的友好性。这里在一个 Activity 中同时并排显示两个 Fragment，见图 3.1 左图。如果用户单击左边 Fragment 文章列表中的一项，选中的文章内容直接显示在右边的 Fragment B 中，不需要在启动另一个 Activity，也使 App 的显示界面更友好美观。



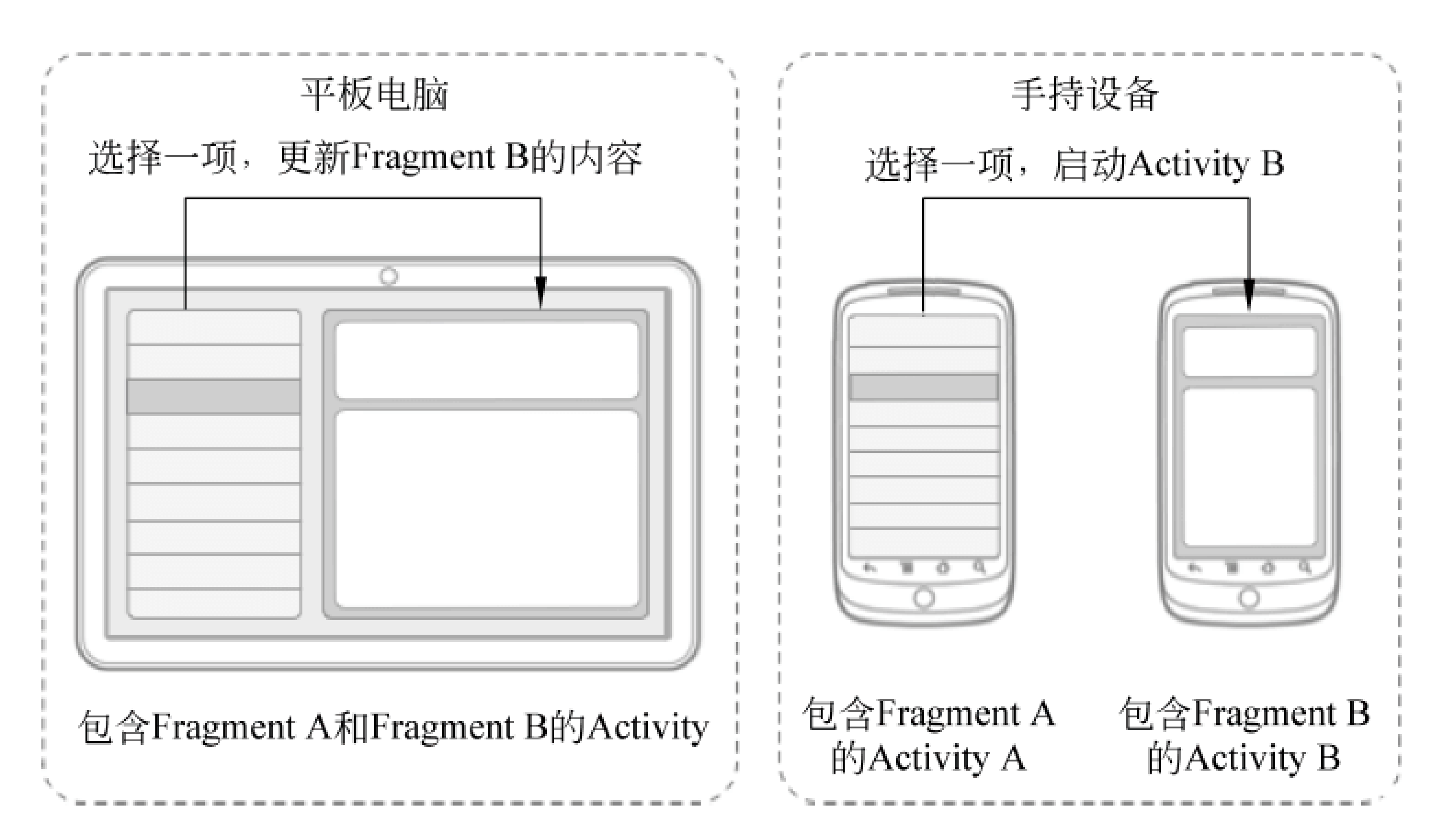


图 3.1 Fragment 的使用实例

Fragment 必须放置在 Activity 中使用，不能独立存在。Fragment 具有自己的生命周期，但它的生命周期也直接受所在的 Activity 生命周期的影响，例如当一个 Activity 被销毁时，所包含的所有 Fragment 也都被销毁了。一个 Activity 中可以包含多个 Fragment，每个 Fragment 都定义自己的布局，并在生命周期中的回调方法中定义自己的动作。Fragment 之间是相互独立的，Activity 通过回退栈来管理每个 Fragment 发生的事件，每一个对 Fragment 操作的事件，都会添加到会退栈中。通过回退栈，允许用户通过回退键取消已经执行的动作。

Activity 的生命周期，直接影响 Fragment 的生命周期。在 Activity 状态转换时，生命周期中每一个回调方法的调用，都导致其中的 Fragment 对应的回调方法的调用。与 Activity 类似，Fragment 以三个状态存在：Resumed、Paused 和 Stopped。在状态之间转换时，系统会调用相应的回调方法。下面以一个应用实例，来详述 Fragment 的创建和使用(见图 3.2)。

Fragment 状态之间转换时，会有多个事件发生，每一个事件都会有 Fragment 回调方法调用。这些事件对应的回调方法如下：

- onAttach()在 Fragment 对象添加到 Activity 中时调用。
- onCreate()在创建 Fragment 对象时调用。
- onCreateView()在 Fragment 绘制用户界面时调用。
- onActivityCreated()所在 Activity 和 Fragment 的 UI 界面创建时调用。
- onStart()在任何 UI 变化时，Fragment 开始变为可视状态时调用。
- onResume()在 Fragment 开始变为运行状态时调用。
- onPause()在 Fragment 运行状态结束，线程挂起，所在 Activity 不再是前台界面时调用。
- onSaveInstanceState()在 Fragment 运行状态结束，保存 UI 状态时调用时。
- onStop()在可见状态结束时调用。
- onDestroyView()在 Fragment 视图被删除时调用。

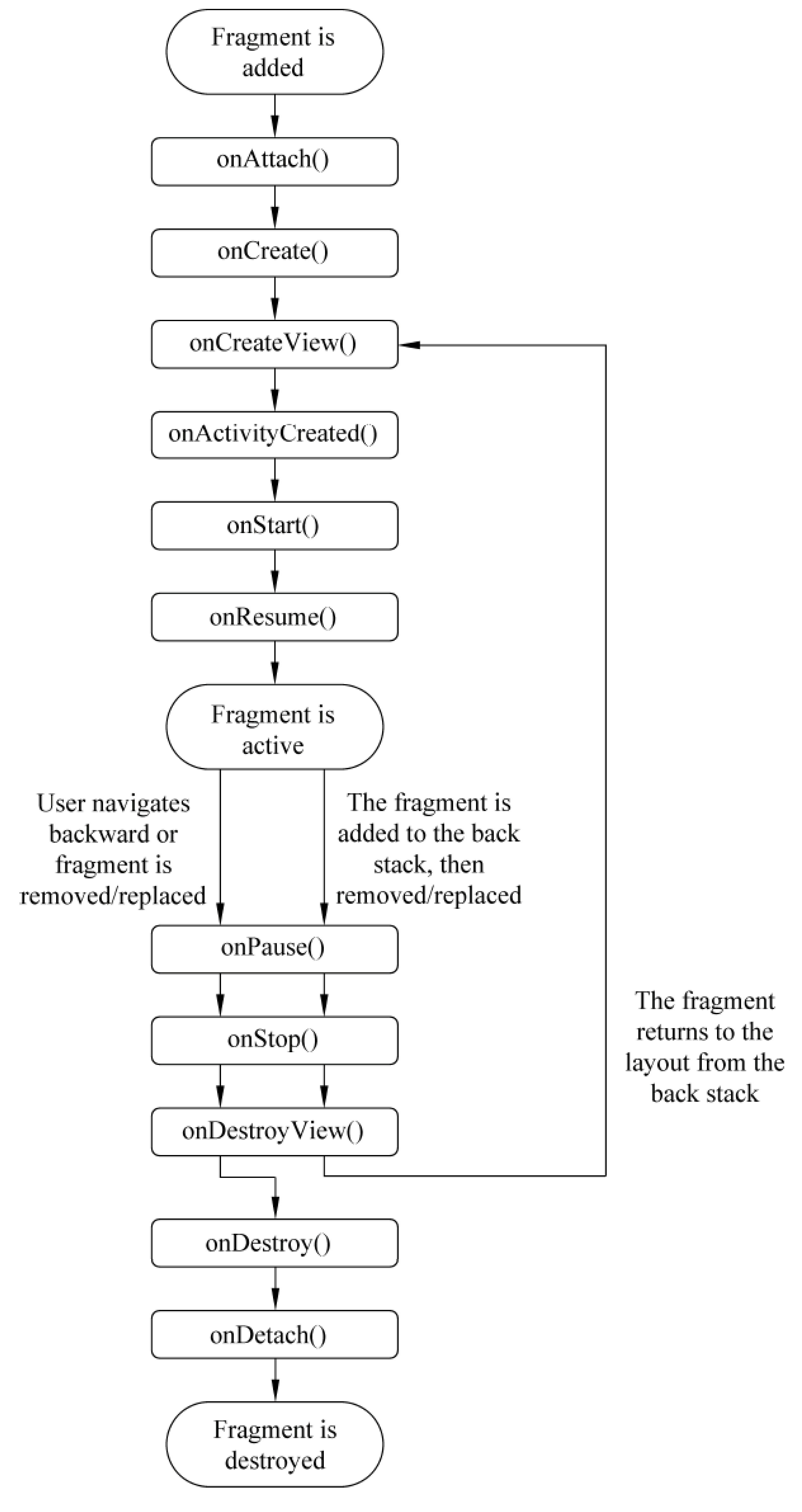


图 3.2 Activity Fragment 的生命周期

- onDestroy()在 Fragment 生命周期结束时调用。
- onDetach()当 Fragment 被所在 Activity 删除时调用。

Activity 中的每一个 Fragment 都是 Fragment 类的子类。在定义 Fragment 时，除了可以直接使用 Fragment 类，还有一些子类可以使用。例如，DialogFragment，直接定义浮动显示的对话框；ListFragment，已经定义好了一个列表的显示；PreferenceFragment，可以把偏好对象的系列显示成一个列表。

在进行 UI 布局设计时，考虑把每一个 Fragment 都设计成 Activity 的可重用组件。

这样，一个设计好的 Fragment 不仅可以在多种屏幕设计时使用，也可以由同一设备的多个 Activity 使用。在构建 Activity 时，既可以静态使用多个 Fragment，也可以在 Activity 运行时根据用户的交互情况，对 Fragment 进行添加、移除、替换以及执行其他动作。

下面详述 Fragment 的创建和使用。

3.1.2　创建和使用 Fragment

在 Activity 中使用 Fragment，可以通过 XML 布局文件先定义 Fragment 的 UI 布局，然后直接在 Activity 的 XML 布局文件中引用定义好的 Fragment。通过这种方式在 Activity 使用 Fragment 的方法，称为在 Activity 中静态添加 Fragment。

在 Activity 中静态添加 Fragment 需要完成下面几项工作：

(1) 创建 Activity 子类和 Fragment 子类。

(2) 定义 Fragment 布局。

(3) 添加 Fragment 布局到 Activity 布局中。

(4) Fragment 中事件处理。

(5) 在 Manifest 文件中注册 Activity。

下面为了讨论静态添加 Fragment 的过程，在 Android Studio 建立一个新的 App 项目，命名为 C03Fragment，主 Activity 命名为 MainActivity，Company Domain 定义为 uibe. edu. cn，实现如图 3. 3 所示的运行显示效果。在这个例子用户界面的 Activity 中，设置了两个 Fragment，其中一个 Fragment 显示图片列表，另一个显示大图片和图片描述，并实现了单击列表中图标在另一个 Fragment 中显示图片和描述的事件处理。



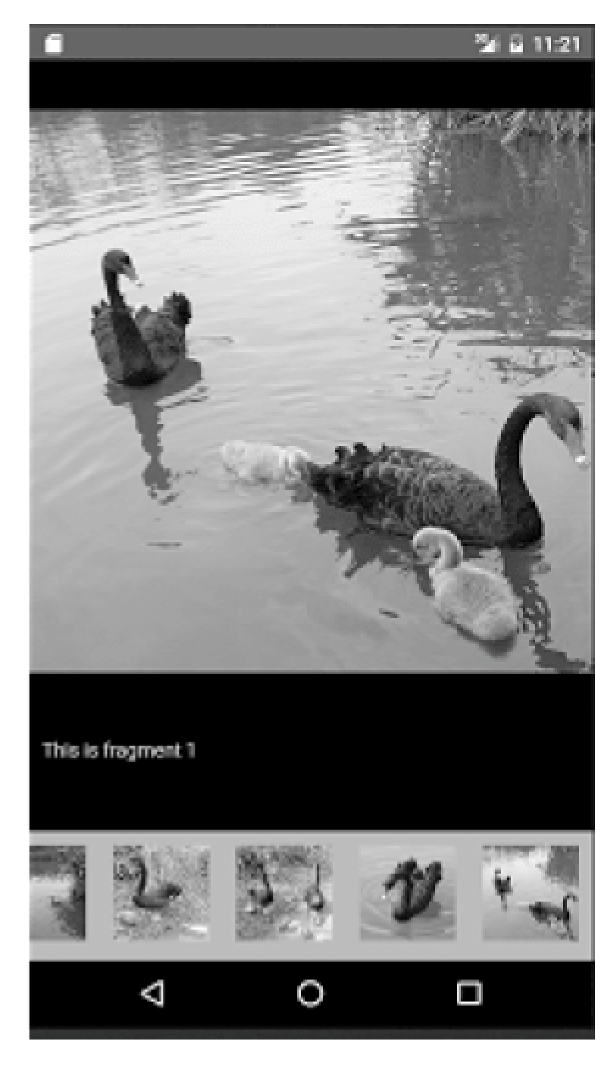


图 3. 3　Fragment 实例效果

下面通过这个例子，详述静态添加 Fragment 具体的步骤和代码。

1. 创建 Activity 和 Fragment 子类

在这个例子中，在 MainActivity 中显示两个 Fragment 的 UI 定义内容。首先打开

MainActivity，将其继承的父类改为 FragmentActivity，见代码 3.1。在修改完成后，按 Alt+Enter 键，导入所需的 java 包。

代码 3.1　修改 MainActivity 类

```
public class MainActivity extends FragmentActivity {
    …
}
```

然后，创建 Fragment 子类。在 Android Studio 的 Project 视图中，右击 app，选择 New→Fragment→Fragment(Blank)，在 Configure Component 对话框中，设置 Fragment Name 为 PictureFragment，选定默认的 Fragment Layout Name，取消选中其下面的两个复选框，单击 Finish 按钮。用同样的步骤创建另一个 TinyFragment 类，见图 3.4。

Fragment 与 Activity 有类似的周期，在创建 Fragment 时，如果没有取消选中图 3.4 中的复选框，Fragment 的类中会列出所有生命周期的回调函数。

在这个例子中，PictureFragment 只需要在 onCreateVeiw()中，使用下面的代码获取 XML 布局文件定义的 Fragment 布局就可以了。后面可以根据需要，在必要的生命周期的回调函数中添加动作。

```
inflater.inflate(R.layout.fragment_picture, container, false);
```

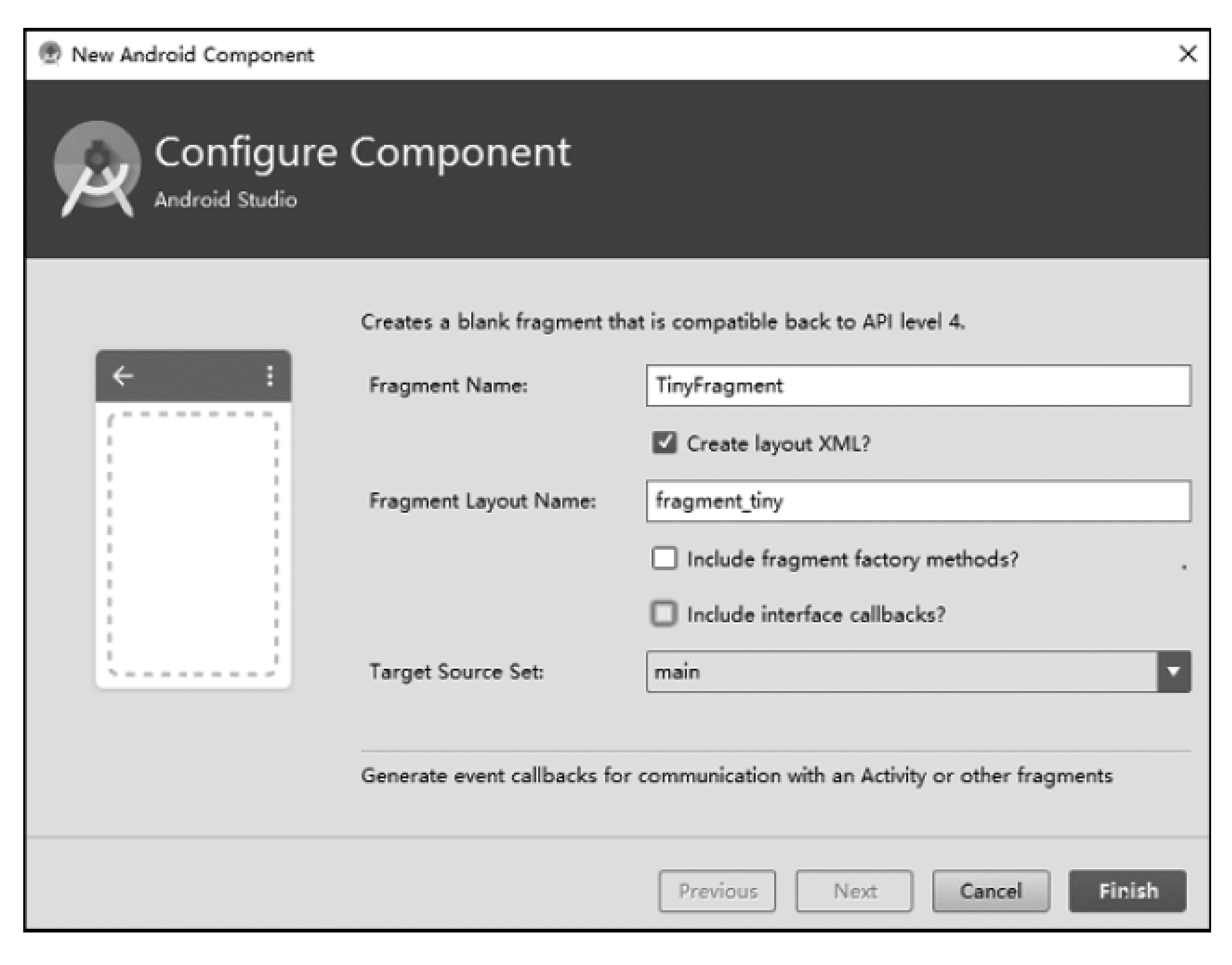
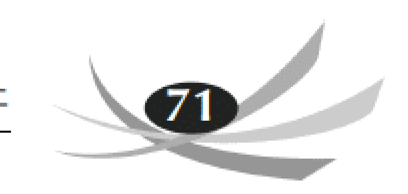


图 3.4　创建 Fragment

2. 定义 Fragment 布局

Fragment 的 XML 布局定义与 Activity 类似，都是在某个布局内放置布局和组件来定义 UI 的显示层次和结构。这里，首先把要显示的图片复制到 res/drawable 目录下，形成图片资源文件，然后打开 res/layout 目录下 PictureFragment 的布局文件 fragment_pciture.xml，添加 UI 界面需要的 ImageView 和 TextView，各属性及其他具体设置等完整的代码见代码 3.2。其中 android:weight 表示组件在布局中显示时所占的比重，也就



是比例大小。

代码 3.2　fragment_picture.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:background="@android:color/background_dark">

        <ImageView
            android:id="@+id/imageView_show"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="3"
            android:contentDescription="@string/swan"
            android:src="@drawable/swan1_1">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="100dp"
            android:layout_weight="1"
            android:text="This is fragment 1"
            android:layout_margin="10dp"
            android:textColor="@android:color/holo_green_light" />

</LinearLayout>
```

同样打开 res/layout 目录下 TinyFragment 的布局文件 fragment_tiny.xml，使用水平滚动视图 HorizontalScrollView，添加图标的滚动功能，并使用一个 ImageView 定义需要显示的一个小图标，各属性及其他具体设置等完整的代码见代码 3.3。如果要在 Fragment 中添加更多的 ImageView，可以按代码 3.3 中的 ImageView 控件的设置，在添加的 LinearLayout 中，注意设置每个 ImageView 不同的 ID 值。

在本章的例子中，共添加了六个 ImageView，以方便看到滑动的效果。

代码 3.3　fragment_tiny.xml

```
<HorizontalScrollView xmlns:android="http://schemas.android.com/apk/res/
android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/scrollView"
    android:background="@android:color/darker_gray">
```

```
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:background="@android:color/darker_gray">

        <ImageView
            android:id="@+id/tinyImage1"
            android:layout_width="70dp"
            android:layout_height="70dp"
            android:contentDescription="@string/swan"
            android:src="@drawable/swan1"
           android:layout_margin="10dp" />

    </LinearLayout>
</HorizontalScrollView>
```

3. 添加到 Activity 中

每一个 Fragment 都必须放置在一个 FragmentActivity 中，FragmentActivity 是 Android 的 Support Library 中专门用于 Fragment 处理的 Activity 子类，从 API level 11 开始支持。如果是低于 API level 11 版本的 Android 系统，就直接使用 Activity。

在 FragmentActivity 的 XML 布局文件中直接定义所使用的 Fragment，以及它们在 Activity 中的层次和结构，完成 Fragment 与这个 FragmentActivity 的关联过程，就是静态添加 Fragment 的过程。

在这个例子中，MainActivity 的 XML 布局文件 activity_mian. xml 中定义了界面所需要使用的两个 Fragment：PictureFragment 和 TinyFragment，以及它们的层次结构和属性，完整代码见代码 3.4。

代码 3.4　activity_main. xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <fragment
        android:id="@+id/fragment1"
        android:name="cn.edu.uibe.c03fragment.PictureFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="6"
```

```
        tools:layout="@layout/fragment_picture" />

    <fragment
        android:id="@+id/fragment2"
        android:name="cn.edu.uibe.c03fragment.TinyFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        tools:layout="@layout/fragment_tiny" />

</LinearLayout>
```

然后可以在 MainActivity 的 onCreate()方法中使用“setContentView(R. layout. activity_main);”调用定义的布局文件，生成用户界面。这时，还没有实现单击小图标在大图中显示的功能。要实现这一步，还需要对 TinyFragment 中的 ImageView 进行单击事件处理。

4. Fragment 中组件的事件处理

相对于 Actiivty 来说，Fragment 中组件的事件处理比较复杂。因为在目前的 Android 版本中，不支持 Fragment 中使用 android:onClick 属性设置单击事件回调方法，也就是说，不能在 Fragment 的布局文件中设置 ImageView 的单击响应方法，只能通过基于监听接口的事件处理机制来进行处理，通过 Java 源程序实现 onClickedListener 来进行事件处理。

在通过 View 的 onClickedListener 处理 Fragment 时，只有静态添加的 Fragment 中的控件，才可以使用 Activity 的 findViewById()方法，通过 R. id 来获取 Fragment 中的控件对象。如果是动态添加的 Fragment，无法使用这种方式来获取对象，只能采用其他的方式。

代码 3.5 中的 showImage()方法中，实现了 TinyFragment 中的 ImageView 单击事件处理。showImage()方法使用 Activity 的 findViewById()方法获取了 TinyFragment 中的 ImageView 对象，并把 onClickedListener 注册到这个控件，在其 onClick()方法中实现单击事件响应，把单击的图片显示到 PictureFragment 的大图中。

到此为止，就完成了本节开始所显示例子的功能。

代码 3.5　TinyFragment 中的 ImageView 单击事件处理

```
package cn.edu.uibe.c03fragment;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.view.View;
import android.widget.ImageView;

public class MainActivity extends FragmentActivity {
```

```
    private ImageView showImg;
    private ImageView tinyImg;
    int imgId[]={R.id.tinyImage1,R.id.tinyImage2,R.id.tinyImage3,R.id.
    tinyImage4,R.id.tinyImage5,R.id.tinyImage6};

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        showImg=(ImageView)findViewById(R.id.imageView_show);
        showImage();
    }

    protected void showImage(){
        for(int i=0;i<imgId.length;i++) {
            tinyImg=(ImageView)findViewById(imgId[i]);
            tinyImg.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    ImageView img=(ImageView) v;
                    showImg.setImageDrawable(img.getDrawable());
                }
            });
        }
    }
}
```

3.1.3 创建动态 UI

Android 系统支持 Fragment 的主要目的是为了让 App 适应不同的屏幕尺寸和位置，方便 App 的 UI 设计和代码在不同类型的屏幕上执行时重用。因此根据屏幕不同的尺寸和位置，动态改变用户界面显示的内容，让用户界面美观、友好、适用，是 Fragment 更重要的作用。

所谓动态 UI，是在 Activity 运行时 App 能够根据屏幕的状态自动添加、删除、替换组成用户界面的 Fragment 或 Fragment 中的组件，能够自动重新组织 Fragment 的布局，给用户提供一个自适应的操作显示界面。

这种在 Activity 的生命周期中，对 Fragment 进行动态添加、删除、替换或其他动作的行为，称为 Fragment 事务。Android 系统使用 FragmentManager 创建 FragmentTrasaction 来完成 Fragment 事务处理。

在进行 Fragment 事务处理时，初始添加 Fragment 的事务必须在 Activity 的 onCreate()方法中完成，而且这个 Activity 在需要添加 Fragment 的位置必须包含一个容器类的 View，例如 LinearLayout。这个容器类的 View 可以在布局文件中静态定义，也

可以在 Activity 的 onCreate()中由 Java 源程序预先创建,添加到 Activity 中。

如果创建动态 UI,需要完成下面的工作:

(1) 创建 Activity 和 Fragment 子类。

(2) 定义需静态或动态添加的 Fragment 布局。

(3) 根据添加 Fragment 和 Fragment 事务的策略,定义 Activity 布局。

(4) 动态添加、删除或替换 Fragment。

(5) 动态 Fragment 的事件处理。

(6) 在 Manifest 文件中声明 Activity。

为了详细讨论如何在 Activity 上实现 Fragment 事务,如何处理动态 Fragment 中组件的时间处理,下面我们在上 3.1.2 节 C03Fragment 项目例子的基础上,修改程序,实现在竖屏时动态添加 TinyFragment,在横屏时动态删除 TinyFragment,以及在 TinyFragment 中的 ImageView 单击事件处理,并在实现这个例子的过程中,详细描述具体的步骤和代码。

1. 定义 Activity 布局

因为是在 C03Fragment 项目的基础上对程序进行修改,所以创建动态 UI 的(1)、(2)步都已经完成。下一步需要修改 MainActivity 的布局文件 activity_main.xml,删除 TinyFragment 的定义部分,取消 MainActivity 在初始化时静态添加 TinyFragment,在需要动态添加 TinyFragment 的位置定义一个容器 LinearLayout,容纳在 MainActivity 运行时动态添加的 TinyFragment,完成第(3)步。

添加的容器 LinearLayout 的具体设置和代码见代码 3.6。

代码 3.6　动态添加 Fragment 的 Activity 布局定义

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <fragment
        android:id="@+id/fragment_picture"
        … />

    <LinearLayout
        android:id="@+id/content_tiny"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:background="@android:color/background_dark"
```

```
        android:orientation="horizontal">
    </LinearLayout>

</LinearLayout>
```

2. 动态添加和删除 Fragment

Activity 实现动态添加、删除和替换等 Fragment 事务，是通过类 FragmentManager 和类 FragmentTrasaction 来实现的，见代码 3.7。

代码 3.7　Fragment 事务处理

```
FragmentManager manager=getSupportFragmentManager();                //(1)
FragmentTransaction transaction=beginTransaction();                 //(2)

transaction.add(R.id.fragment_container, firstFragment);            //(3)
transaction.remove(R.id.fragment_container, secondFragment);        //(3)
transaction.replace(R.id.fragment_container, thirdFragment);        //(3)

transaction.commit();                                               //(4)
```

一般来说，实现 Fragment 事务需要下面几步：

(1) 调用 Activity 的 getSupportFragmentManager()方法，获取当前 Activity 的 FragmentManager 对象，赋值给 FragmentManager 变量 manager。

(2) 通过对象 manager 调用 beginTransaction()，创建启动 Fragment 事务对象，并赋值给一个 FragmentTransaction 变量 transaction。

(3) 通过对象 transaction 调用 add()、remove()、replace()执行 Fragment 事务，动态添加、删除、替换 Fragment。在执行 Fragment 事务时，可以根据需要，执行一个或多个事务。

(4) 使用对象 transaction 调用 commit()方法提交所有的 Fragment 事物。

在本例中，修改 MainActiivity 的 onCreate()方法，在其中添加 Fragment 事务的语句。通过 if 语句判断当前屏幕的状态，在竖屏时，实现 TinyFragment 的动态添加，在横屏时实现 TinyFragment 删除，见代码 3.8。

代码 3.8　添加和删除 TinyFragment

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    showImg=(ImageView)findViewById(R.id.imageView_show);
    txt=(TextView)findViewById(R.id.imageTxt);

    if(this.getResources().getConfiguration().orientation==
    Configuration.ORIENTATION_PORTRAIT){
```

```
        tiny=new TinyFragment();
        tiny.setArguments(getIntent().getExtras());

        FragmentManager manager=getSupportFragmentManager();
        FragmentTransaction transaction=manager.beginTransaction();
        transaction.add(R.id.content_tiny,tiny,"tinyfragment");

        transaction.commit();
    }else{
        FragmentManager manager=getSupportFragmentManager();
        tiny=manager.findFragmentByTag("tinyfragment");
        FragmentTransaction transaction=manager.beginTransaction();
        if(tiny!=null) transaction.remove(tiny);
        transaction.commit();
    }
}
```

这时,运行 App,会发现 App 出现运行异常。这是因为静态 Fragment 中的控件,可以直接通过 Activity 的 findViewById()方法获取,设置 onClickedListener,但在动态 Fragment 中,无法通过同样的方式获取控件,得到的只是空指针。对空指针进行操作,App 就出现了异常。所以对于动态 Fragment 中的控件事件处理,需要另一套机制。

在 MainActivity 中,删除原有的事件处理代码,即删除 showImage()方法实现,并删除的 onCreate()中的 showImage()调用语句。重新运行 App,就可以看到预期的显示效果了:竖屏时显示 TinyFragment 中的小图标列表;横屏时,只能看到 PictureFragment 中的大图和注释信息,见图 3.5 所示的动态 Fragment 显示效果。

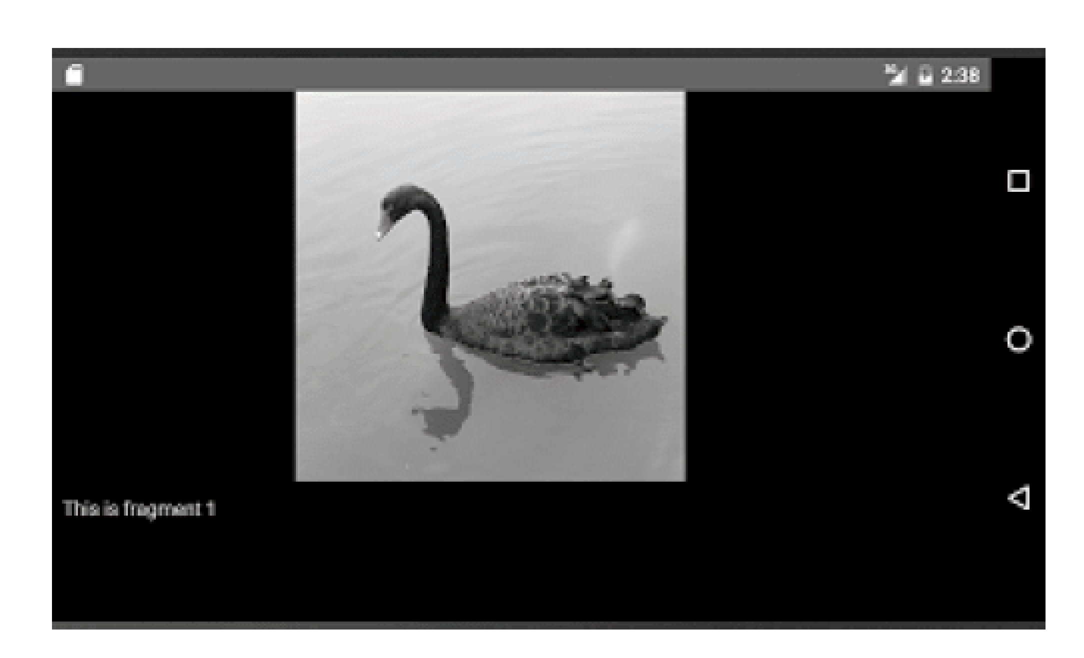


图 3.5　动态 Fragment 显示效果

3. 动态 Fragment 的事件处理

在动态用户图形界面上,经常会通过一个 Fragment 中控件的操作,来影响另一个 Fragment 的显示结果。例如,单击邮件列表中的标题,会改变邮件内容的显示。要实现这个功能,程序员需要解决两个问题:

- 在 Fragment 之间传递消息。
- 确定事件源,设定事件处理器。

在 Android 系统中，所有的 Fragment 之间进行通信，必须通过其所在的 Activity 进行，两个 Fragment 之间无法进行直接的消息传递。

要实现一个 Fragment 与其所在的 Activity 之间的通信，可以在这个 Fragment 中定义一个 Interface 接口，然后再 Activity 中实现其包含的方法。当 Fragment 运行设置事件源的回调方法 onAttach()中时，会捕获这个接口，实现与 Activity 的通信。

因为在 Activity 中无法获取动态 Fragment 中的控件，Fragment 也无法在 XML 布局文件中设置事件响应回调方法，所以必须使用基于监听器的事件处理机制，在 Fragment 生命周期回调方法 onAttach()中，通过 Fragment 的 findViewById()获取需要进行事件处理的控件，在控件上设置监听器，在事件处理方法中编写事件处理代码。当然在 onCreate ()中实现也可以。

本例中要实现单击 TinyFragment 中的小图标，在 PictureFragment 中显示对应的大图，必须把 TinyFragment 中单击的信息传递给 PcitureFragment，然后才能使 PictureFragment 中的 ImageView 显示出对应的大图。下面通过例子，说明如何使用接口来传递消息，进行事件处理。

1）定义 Interface

首先，在 TinyFragment 中定义接口 ImageClickedListener，并声明此接口的变量 mCallback 作为 TinyFragment 和 Activity 传递消息的工具，见代码 3.9。

代码 3.9　定义 ImageClickedListener 接口

```
public class TinyFragment extends Fragment {
    ImageClickedListener mCallback;

    public interface ImageClickedListener{
        public void updateImage(ImageView img);
    }
```

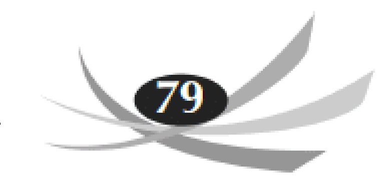
然后在 TinyFragment 的 onAttach()方法中，通过 getView(). findViewById()获取 ImageView 对象，在其上设置 setOnClickListener，并在其事件处理方法 onClick()中使用 mCallback 调用 MainActivity 定义的 updateImage()方法，通过参数把消息传递给 MainActiivty，并通过 MainActivity 对 PictureFragment 进行操作，见代码 3.10。

代码 3.10　定义 ImageClickedListener

```
public class TinyFragment extends Fragment {
    ImageView tinyImage;
    int imgId[]={R.id.tinyImage1,R.id.tinyImage2,R.id.tinyImage3,R.id.
    tinyImage4,R.id.tinyImage5,R.id.tinyImage6};
    int tinysize=6;

    ...

    public void onAttach(Activity activity) {
```




```
        super.onAttach(activity);

        try {
            mCallback=(ImageClickedListener) getActivity();
        } catch(ClassCastException e) {
            throw new ClassCastException(getActivity().toString()
                    +" must implement ImageClickedListener ");
        }

        for(int i=0;i<imgId.length;i++){
            tinyImage=(ImageView)getView().findViewById(imgId[i]);
            if(tinyImage!=null)
                tinyImage.setOnClickListener(new View.OnClickListener(){
                    @Override
                    public void onClick(View view) {
                        mCallback.updateImage((ImageView)view);
                    }
                });
        }
    }
}
```

2）实现 Interface

TinyFragment 与 MainActivity 之间消息能传递成功的关键，在于 TinyFragment 中把 mCallback 获取的 MainActivity 作为接口使用，调用了接口 ImageClickedListener 在 MainActivity 中实现后的 updateImage()方法。具体的实现代码见代码 3.11。

代码 3.11　在 MainActivity 中实现 ImageClickedListener

```
public class MainActivity extends FragmentActivity implements
TinyFragment.ImageClickedListener {
    …
    public void updateImage(ImageView img){
        showImg.setImageDrawable(img.getDrawable());
        txt.setText(showImg.getContentDescription());
    }
}
```

到此为止，就完成了创建动态 UI 的例子，可以运行 App 查看运行效果。

3.2　常用基本控件

Android 系统为用户界面的设计提供了很多图形控件，在这一节中我们介绍一些常用的基本控件，及其事件处理方法。

3.2.1 事件处理机制

由于 Android 应用程序使用 Java 程序编写，在应用程序运行中，用户对移动终端键盘、屏幕和位置的操作都转化成事件对象，Android 系统通过对这些事件的捕获后，执行相应的处理代码，实现与用户的交互，完成预定的功能。这个过程就是 Android 的事件处理。

Android 的事件处理机制有两种：基于监听接口和基于回调。这两种机制的原理和实现方法都有所不同。

1. 基于监听接口的事件处理机制

事件机制是处理事件的方式和方法。Android 的基于监听接口的事件处理机制，完全采用了 Java 的事件处理机制。

Java 的事件处理基于委托事件处理模型，把事件的发生与事件的处理相分离，由监听器监听（等待）事件发生，事件发生后再由监听器委托事件处理器处理。Java 采取了授权事件模型（Delegation Event Model），事件源可以把在其自身所有可能发生的事件分别授权给不同的事件处理者来处理。

在事件处理的过程中，主要涉及三个主要部分：事件源、事件和事件处理。

1）事件源（Event Source）

事件源，是指触摸屏、键盘或位置传感器操作针对的控件或容器。事件发生时，也就是出现某个控件被触摸操作或移动终端位置移动，这个控件，也就是事件源类负责发送事件发生的通知，并通过事件源查找自己的事件监听者队列，并将事件信息通知队列中的监听者来完成。同时，事件源还在得到有关监听者信息时负责维护自己的监听者队列。

2）事件（Event）

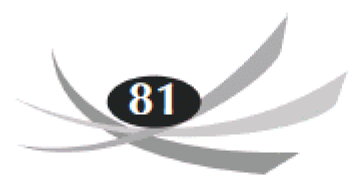
事件，是指对组件或容器的触摸屏、键盘或位置的一个操作，用类描述。例如键盘事件类 KeyEvent 描述键盘事件的所有信息：键按下、释放、双击、组合键以及键码等相关键的信息。

3）事件处理（Event Handler）

Java 的事件处理由事件监听器类和事件监听器接口来实现。事件发生后，事件源将相关的信息通知对应的监听器，事件源和监听者之间通过监听者接口完成这类的信息交换。事件监听者类就是事件监听者接口的具体实现，事件发生后，该主体负责进行相关的事件处理，同时，它还负责通知相关的事件源，自己关注它的特定的事件，以便事件源在事件发生时能够通知该主体。

外部的操作，例如按下按键、触摸屏幕单击按钮或转动移动终端等动作，会触发事件源上的事件。对于单击按钮的操作来说，事件源就是按钮，它会根据这个操作生成一个按钮按下的事件对象，这对于系统来说，就产生了一个事件。

事件的产生会触发事件监听器，事件本身作为参数传入到事件处理器中。事件监听器是在通过代码在程序初始化时注册到事件源的，也就是说，在按钮上设置一个可以监听按钮操作的监听器，并且通过这个监听器调用事件处理器，事件处理器针对这个事件编写代码，例如弹出一条信息。



在代码实现时，基于监听器的事件处理都需要做三项工作：

• 定义监听器类，覆盖对应的抽象方法，在监听器中针对事件编写响应的处理代码。
• 创建监听器对象。
• 注册监听器。

2. 基于回调的事件处理机制

Android 的另一种事件处理机制是回调机制。

通常情况下，程序员写程序时，需要使用系统工具类提供的方法来完成某种功能，例如调用 Math.sqrt()求取平方根。但是，某种情况下系统会反过来调用一些类的方法，例如对于用作组件或插件的类则需要编写一些供系统去调用的方法，这些专门用于被系统调用的方法被称为回调方法，也就是回过来系统调用的方法。

Android 平台中，每个 View 都有自己的处理事件的回调方法，开发人员可以通过重写 View 中的这些回调方法来实现需要的响应事件。当某个事件没有被任何一个 View 处理时，便会调用 Activity 中相应的回调方法。例如，有一个按钮按下的事件发生了，但编码过程中这个按钮并没有对这个事件做任何处理，它所在的 Activity 中的任何组件也并没有对这个事件做任何处理，这时系统会调用 Activity 相应的回调方法 onKeyDown()。

回调机制实质就是将事件的处理绑定在组件上，由 GUI 组件自己处理事件，回调机制需要自定义 View 来实现，自定义 View 重写该 View 的事件处理方法就可以了。例如，Activity 和 Fragment 的生命周期中的各种状态发生变化时，调用的 onResume()等方法，也是回调方法。

Android 也提供自定义回调方法，例如在前面两章的例子中，通过 XML 文件中的组件 android:onClicked 属性，自定义了回调方法的名称，然后直接在 Activity 中覆盖这个方法，进行单击事件处理。这是一种非常有效的简单的事件处理实现方式。

这两种事件处理机制都是在 Android 应用程序开发过程中常用的机制，程序员可以根据实际情况，选择合适的方式。前面的章节中，我们既运用了基于监听接口的事件处理机制，例如在 3.1 节中，也运用了基于回调的事件处理机制，例如在第 1、2 章的例子中。在以后的章节中，我们也会根据实际情况，选择合适的事件处理机制。

3.2.2 按钮控件

Android 提供的按钮控件有很多种，包括基本的 Button、RadioButton、CheckBox、ToggleButton 和 Switch 都是按钮的类型。

Button 类控件继承自 TextView，因此也具有 TextView 的宽和高设置、文字显示等一些基本属性。Button 类控件在应用程序中的定义，与其他图形控件一样，一般都在布局文件中进行定义、设置和布局设计。setText()和 getText()是 Button 类控件最常用的方法，用于设置和获取 Button 显示的文本。

Button 类控件一般会与单击事件联系在一起。对于基本的 Button，可以采用两种方式处理单击事件。一种使用 Button 的 setOnClickListener()方法为其设置 OnClickListener，把具体的事件处理代码写在 onClick(View v) 方法中；另一种在 XML 布局文件中，使用 Android:OnClick 属性为 Button 指定单击事件发生时执行的方法。

如果在 XML 布局文件中,使用 Android:OnClick 属性指定了单击事件的回调方法,这个方法在 Java 应用程序中必须是 public 的,而且只有一个 View 类型的参数。

在按钮类控件的使用过程中,属性设置和事件处理稍有不同,下面具体说明各按钮类控件如何对事件进行处理。在具体调试运行过程中,创建资源文件和 Activity 的具体步骤与前面例子相同,请参考其编写完整的代码,运行并查看效果。

1. Button

Button 即按钮。按钮控件可以有文本或者图标,也可以文本和图标同时存在(见图 3.6),当用户触摸时就会触发事件。



图 3.6 各种按钮

根据按钮控件的组成方式,创建按钮控件有三种方式:

1) 如果由文本组成,使用 Button 类创建

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    … />
```

2) 如果由图标组成,使用 ImageButton 类创建

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon"
    … />
```

3) 如果文本和图标都有,使用 Button 类的 android:drawableLeft 属性

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:drawableLeft="@drawable/button_icon"
    … />
```

除了按钮上的文本和图标,按钮的外观(如背景图片和字体)可能会因为设备或者 Android 版本的不同而有所不同,随着 Android 版本的升级,其界面的样式也发生变化,而厂家的也会定制输入控件的默认样式。

如果要控制控件,使用适用于整个应用程序的样式。例如,要确保所有运行 Android 4.0 甚至更高版本的设备在应用程序使用 Holo 主题,需要在 manifest 的元素中声明 android:theme="@android:style/Theme.Holo"。

在 XML 布局文件中,可以使用 Button 的一些属性来定义按钮的外观。定制不同的

背景，可以指定<android：background>属性为绘图或颜色的资源，也可以是自定义的背景。其他的属性，如字体、大小、边框等，可以参照 TextView 和 View 的 XML 属性。详细的 XML 属性说明可以从链接 http://developer.android.com/reference/android/R.styleable.html 查阅。

下面是一个简单的例子，使用了一种无边框按钮。无边框按钮与基本按钮相似，但是无边框按钮没有边框或背景，但在不同状态如单击时会改变外观。要创建一个无边框按钮，为按钮应用<borderlessButtonStyle>样式，见代码 3.12。

代码 3.12　按钮外观设置

```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage"
        style="?android:attr/borderlessButtonStyle" />
```

2. RadioButton

RadioButton 就是单选按钮，在 Android 开发中应用的非常广泛。RadioButton 的外形是单个圆形的单选框，具有选中或未选中两种状态。在 RadioButton 没有被选中时，用户能够按下或单击来选中它。与复选框不同的是，用户一旦选中就不能够取消选中。

一般来说，实现 RadioButton 需要由 RadioButton 和 RadioGroup 配合使用。RadioGroup 是单选组合框，是可以容纳多个 RadioButton 的容器。在没有 RadioGroup 的情况下，RadioButton 可以全部都选中；当多个 RadioButton 被 RadioGroup 包含的情况下，RadioButton 只可以选择一个。

RadioButton 的事件处理，可以使用 setOnCheckedChangeListener()方法注册单选按钮的监听器，也可以采用在 XML 布局文件中指定处理方法的方式。

下面这个例子，在 XML 布局文件中定义了一个具有四个 RadioButton 的 RadioGroup，一个文本显示框 TextView 控件和一个按钮 Button 控件，见代码 3.13。当一个 RadioButton 被选中时，在 TextView 控件中显示选择项的文本，如果单击按钮，将清除选中的项目。

代码 3.13　radiobutton_layout.xml

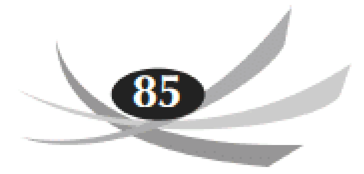
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <RadioGroup
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:checkedButton="@+id/lunch"
        android:id="@+id/menu">
        <RadioButton
            android:text="@string/radio_group_1_breakfast"
            android:id="@+id/breakfast"
            />
        <RadioButton
            android:text="@string/radio_group_1_lunch"
            android:id="@id/lunch" />
        <RadioButton
            android:text="@string/radio_group_1_dinner"
            android:id="@+id/dinner" />
        <RadioButton
            android:text="@string/radio_group_1_all"
            android:id="@+id/all" />
        <TextView
            android:text="@string/radio_group_1_selection"
            android:id="@+id/choice" />
    </RadioGroup>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radio_group_1_clear"
        android:id="@+id/clear" />
</LinearLayout>
```

在这个例子中没有指定事件处理的方法，因此，在 Java 应用程序中，采用控件相对应的两个事件监听器 RadioGroup. OnCheckedChangeListener 和 View. OnClickListener 来处理对 RadioGroup 和 RadioButton 的事件，具体的事件处理代码写在 onCheckedChanged()和 onClick()接口方法中，分别实现根据选项更新 TextView 的显示和清除 RadioButton 选中的功能，见代码 3.14。

代码 3.14　RadioGroupActivity.java

```
public class RadioGroupActivity extends Activity implements
    RadioGroup.OnCheckedChangeListener, View.OnClickListener {
    setContentView(R.layout.radiobutton_layout);
    mRadioGroup=(RadioGroup) findViewById(R.id.menu);
    mRadioGroup.setOnCheckedChangeListener(this);
    Button clearButton=(Button) findViewById(R.id.clear);
    clearButton.setOnClickListener(this);
```




```
    public void onCheckedChanged(RadioGroup group, int checkedId) {
            String selection=getString(R.string.radio_group_selection);
            String none=getString(R.string.radio_group_none);
            mChoice.setText(selection
            +(checkedId==View.NO_ID ? none : checkedId));
            }

    public void onClick(View v) {
            mRadioGroup.clearCheck();
    }
}
```

完成应用程序编码后，同样不要忘记要到 AndroidManifest.xml 中注册才能运行。

从上面的例子可以看出，Android 控件的事件处理方法与一般的 Java 图形界面处理类似，只是控件和监听器有所不同，所采用的事件处理机制和原理以及实现步骤都基本相同。

3. CheckBox

CheckBox 就是复选框，具有选中和未选中两种状态。CheckBox 的外形是矩形框，可以通过单击选中或取消选中。在进行事件处理时，应用程序可以根据是否被选中来进行相应的操作，并且对复选框加载事件监听器，来对控件状态的改变做出响应。

下面这个例子，通过 XML 布局文件在用户界面使用 CheckBox 控件来创建一个复选框，实现当复选框被单击时，弹出一个文本消息显示复选框的当前状态。

(1) 创建 XML 布局文件 checkbox_layout.xml，定义一个 CheckBox 控件，并在其中使用 android:onClick 属性指定事件处理的方法名为 onCheckBoxClicked，见代码 3.15。

代码 3.15　checkbox_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <CheckBox
        android:id="@+id/checkbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onCheckBoxClicked"
        android:text="check it out" />

</LinearLayout>
```

（2）创建新类 CheckBoxActivity，实现 XML 文件中指定的单击 CheckBox 控件后的事件处理方法 onCheckBoxClicked(View v)，见代码 3.16。

代码 3.16　CheckBoxActivity.java

```
public class CheckBoxActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        //TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.checkbox_layout);
    }

    public void onCheckboxClicked(View v) {
        //Perform action on clicks, depending on whether it's now checked
        if(((CheckBox) v).isChecked()) {
            Toast.makeText(CheckBoxActivity.this, "选中",
                    Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(CheckBoxActivity.this, "没选中",
                    Toast.LENGTH_SHORT).show();
        }
    }
}
```

完成应用程序编码后，不要忘记要到 AndroidManifest.xml 中注册才能运行。

从上面的例子可以看出，事件处理方法可以采用与 Button 相同的模式，只是在处理过程中，可以针对 CheckBox 不同的状态进行不同编码，实现不同的功能。

4. ToggleButton 和 Switch

如果设置选项只有两种状态，可以使用开关按钮 ToggleButton（见图 3.7 左图）。Android 4.0（API 级别 14）提供了另外一种叫做 Switch 的开关按钮，这个按钮提供一个滑动控件，可以通过添加 Switch 对象来实现（见图 3.7 右图）。

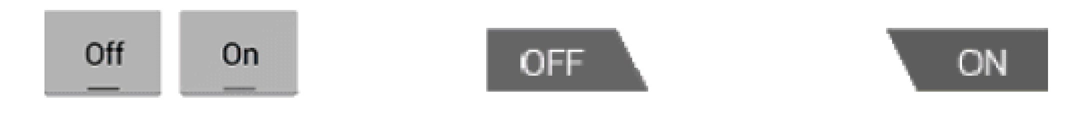


图 3.7　ToggleButton 和 Switch

ToggleButton 和 Switch 控件都是 CompoundButton 组合按钮的子类并且有着相同的功能，所以可以用同样的方法来实现它们的功能。当用户选择 ToggleButtons 和 Switch 时，对象就会接收到相应的单击事件。要定义这个单击事件的响应操作，添加 android:onClick 属性到 XML 布局文件的开关按钮控件中去。例如，代码 3.17 定义了一个 ToggleButton 开关按钮并且设置了 android:onClick 事件单击响应属性。

代码 3.17　在布局文件中定义 ToggleButton

```
<ToggleButton
    android:id="@+id/togglebutton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Vibrate on"
    android:textOff="Vibrate off"
android:onClick="onToggleClicked"/>
```

在这个布局对应的 Activity 里，在 android:onClick 指定的 onToggleClicked()方法中，定义事件处理代码，见代码 3.18。

代码 3.18　onToggleClicked 事件处理

```
public void onToggleClicked(View view) {
    //Is the toggle on?
    boolean on=((ToggleButton) view).isChecked();

    if(on) {
        //Enable vibrate
    } else {
        //Disable vibrate
    }
}
```

与其他图形控件一样，除了在布局文件中定义之外，也可以通过代码的方式，为控件注册一个事件监听器。代码 3.19 中说明了为 ToggleButton 注册监听器的具体实现代码：首先创建一个 CompoundButton. OnCheckedChangeListener 对象，覆盖 OnCheckedChangeListener 接口的抽象方法 onCheckedChanged()，在其中具体实现单击 ToggleButton 对象后的事件处理，然后通过调用此 ToggleButton 对象的 setOnCheckedChangeListener()方法，将监听器绑定到按钮上，见代码 3.19。

代码 3.19　为 ToggleButton 注册事件监听器

```
ToggleButton toggle=(ToggleButton) findViewById(R.id.togglebutton);
toggle.setOnCheckedChangeListener(new CompoundButton.
OnCheckedChangeListener() {
    public void onCheckedChanged(CompoundButton buttonView, boolean
    isChecked) {
        if(isChecked) {
            //The toggle is enabled
        } else {
            //The toggle is disabled
```

```
        }
    }
});
```

完整的应用程序可以参考 CheckBox 和 RadioButton 编写。完成应用程序编码后，同样不要忘记要到 AndroidManifest.xml 中注册才能运行。

3.2.3 Toast 控件

Toast 是在窗口表面弹出的一个简短的小消息，只填充消息所需要的空间，并且用户当前的 Activity 依然保持可见性和交互性。这种通知可自动地淡入淡出，且不接受用户的交互事件。例如，如果用户正在编写一封邮件时，需要接通一个电话，这时界面会弹出一个 Toast 提示，邮件保存为草稿，见图 3.8。

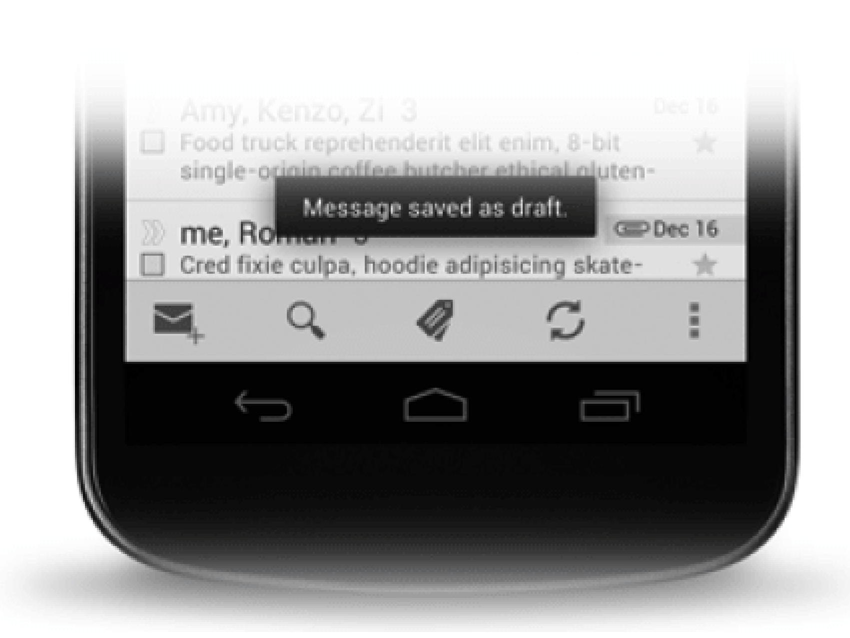


图 3.8　Toast 显示

Toast 是一个在屏幕上显示片刻的提示消息，但是 Toast 不能获得焦点，不能够与用户进行交互。我们可以自定义包括图像的 Toast 布局文件。Toast 通知能够被 Activity 或 Service 创建并显示。如果创建了一个源自 Service 的 Toast 通知，它会显示在当前的 Activity 最上层。如果用户需要对通知做出响应，可以考虑使用 Android 的另一种视图对象状态栏通知(Status Bar Notification)，这将在后面的章节介绍。

如果要使用 Toast，可以直接用 Toast 类的方法 Toast.makeText()实例化一个 Toast 对象。这个方法有三个参数，分别为 Context、要显示的文本消息和 Toast 通知持续显示的时间。Toast.makeText()方法会返回一个按参数设置且被初始化的 Toast 对象，Toast 对象的内容用 show()方法显示。代码 3.20 示例了在 Activity 的 onCreate()方法中如何创建和显示 Toast 信息，在其他视图中实现的代码类似。

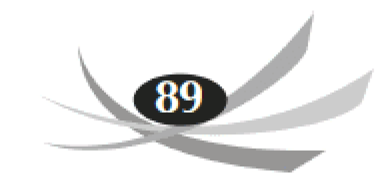
代码 3.20　显示 Toast 通知

```
Context context=getApplicationContext();
CharSequence text="Hello toast!";
int duration=Toast.LENGTH_SHORT;

Toast toast=Toast.makeText(context, text, duration);
toast.show();
```

代码 3.20 中最后两行代码也可以用链式组合方法写且避免创建 Toast 对象，代码如下：

```
Toast.makeText(context, text, duration).show();
```



标准的 Toast 通知水平居中显示在屏幕底部附近。如果要把 Toast 通知放到不同的位置显示，可以使用布局文件来设置 Toast 对象的具体布局，然后在 Activity 加载布局文件后，通过 ID 获取 Toast 对象，使用 show()方法显示其文本消息。

下面我们使用一个简单的例子，来说明如何定义和使用 Toast 自定义的布局文件。要创建一个自定义的布局文件，可以在 XML 布局文件或程序代码中定义一个 View 布局，然后把 View 对象传递给 setView()方法。代码 3.21 中 layout.custom_toast.xml 布局文件是专门为 Toast 对象的布局所做的定义，其中第二行代码 android：id＝"@＋id/toast_layout_root"定义了这个 Toast 布局的 id。这个是一个包含一个图形和一个文本框的布局，其背景、对齐方式和文本颜色也进行了设置。

代码 3.21　Toast 自定义布局

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/toast_layout_root"
              android:orientation="horizontal"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:padding="8dp"
              android:background="#DAAA"
              >
    <ImageView android:src="@drawable/droid"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_marginRight="8dp"
              />
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:textColor="#FFF"
              />
</LinearLayout>
```

Toast 的布局文件创建完成后，需要把这个布局应用到用户界面的 Toast 对象。在应用程序 Activity 的 onCreate()中，首先导入 Activity 的布局资源文件，然后需要使用 LayoutInflater 的对象，通过其 inflate()方法，利用布局文件名和布局的 id 来获取布局文件中定义的布局，下一步使用 Toast 对象的 setView()方法使用这个布局，见代码 3.22。

代码 3.22　使用自定义 Toast 布局

```
LayoutInflater inflater=getLayoutInflater();
View layout=inflater.inflate(R.layout.custom_toast,
                             (ViewGroup) findViewById(R.id.toast_layout_root));

TextView text=(TextView) layout.findViewById(R.id.text);
```

```
text.setText("This is a custom toast");

Toast toast=new Toast(getApplicationContext());
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

除非使用 setView()方法设置自定义布局，否则不要使用公共的 Toast 类构造器。如果不使用自定义的布局，必须使用 makeText(Context，int，int)方法来创建 Toast 对象。

代码 3.22 中的 setGravity(int，int，int)方法，可以重新设置 Toast 对象的显示位置。这个方法有三个参数，分别为 Gravity 常量、X 轴偏移量、Y 轴偏移量。

3.2.4 文本控件

Android 用于文本显示和编辑的控件主要包括 TextView 和 EditText 两种。实际上，Android 的很多控件都继承自 TextView 类，包括 Button、CheckTextView、EditText 等，但用于文本显示时，常用的还是 TextView 和 EditText。因此这里我们主要介绍这两个控件如何定义和使用。

1. TextView

TextView 是 Android 中常用的组件之一，用于显示文字，类似 Java 图形界面里的 Label 标签。TextView 中提供了大量的属性用于设置 TextView 的字体大小、字体颜色、字体样式等。由于很多控件都是 TextView 的子类，它们也继承 TextView 的属性，这给应用程序的界面提供了多种显示组合和样式。

TextView 的属性可以直接在 XML 布局文件中设置，也可以在 Java 应用程序中设置和修改。

例如，用户界面的布局文件 textview_layout.xml 中定义了一个 TextView，我们在 TextView 几个基本属性基础上增加几个属性设置，如 android:textColor="#ff0000"设置字体为红色，android:textSize="24sp"设置字体为 24sp，android:textStyle="bold"设置字体加粗。

如果要在 Java 代码中对 TextView 控件属性进行修改，在其布局文件中必须要给这个 TextView 的 ID 属性赋值。TextView 的 ID 属性是这个 TextView 部件的唯一标识，用于 Java 程序对其进行引用。设定 TextView 的 ID 属性的具体语法如下：

```
android:id="@+id/textview_name"
```

假设在 textview_layout.xml 文件中设定为 android:id="@+id/textvw"，没有增加属性的设置，在 Java 应用程序 TextViewActivity 中可以通过 findViewById()获取 TextView 控件，然后通过对象修改其属性，也可以达到同样的效果，见代码 3.23。

代码 3.23 TextViewActivity.java

```
public class TextViewActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.textview_layout);

        //获取布局中定义的 TextView 组件
        TextView textView= (TextView)findViewById(R.id.textvw);
        //字体设置成红色
        textView.setTextColor(Color.RED);
        //设置成 24sp
        textView.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24f);
        //加粗
        textView.setTypeface(Typeface.defaultFromStyle(Typeface.BOLD));
        //背景设置成黑色
        textView.setBackground(Color.BLACK);
    }
}
```

通过上面的尝试，说明通过 Java 代码程序和 XML 布局文件都可以实现 TextView 属性的设置。不过在 Android 应用系统开发过程中，还是推荐使用 XML 进行布局和界面外观的设计，使用 Java 程序代码实现程序逻辑。

2. EditText

EditText 是 Android 的文本编辑框，是用户和 Android 应用进行数据交互的窗口，可以接受用户的文本数据输入，并将其传送到应用程序中。EditText 是 TextView 的子类，所以 EditText 继承了 TextView 的所有方法和所有属性。

EditText 类似于 Java 图形界面的文本编辑框，但与后者相比，增加从 TextView 继承的属性之后，设置 EditText 的显示和输入时，就可以根据不同的需求设计出更加有个性和特点的交互界面。例如，可以通过 EditText 的属性设置文本编辑框的最大长度、空白提示文字等，或者限制输入的字符类型只能为电话号码。表 3.1 列出了 EditText 常用的一些属性和说明，这些属性也同样适用于 TextView。

表 3.1 EditText 常用的属性

属　　性	说　　明
android:editable	是否可编辑
android:gravity	设置控件显示的位置，默认 top
android:height	设置高度
android:hint	设置 EditText 为空时，文本提示信息内容

续表

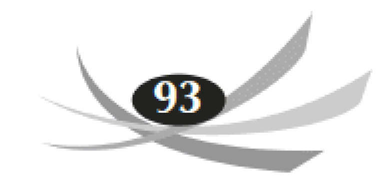
属　　性	说　　明
android:imeOptions	设置附加功能，设置右下角 IME 动作与编辑框相关动作
android:inputType	设置文本的类型，用于帮助输入法显示合适的键盘类型
android:lines	设置 EditText 显示的行数
android:maxLength	设置最大长度
android:maxLines	设置文本的最大显示行数，与 width 或者 layout_width 结合使用，超出部分自动换行，超出行数将不显示
android:numeric	设置为数字输入方式
android:password	以小点“.”显示文本
android:phoneNumber	设置为电话号码的输入方式
android:scrollHorizontally	设置文本超出 TextView 的宽度时，是否出现横拉条
android:textColor	设置文本颜色
android:textColorHighlight	设置被选中后文本颜色，默认为蓝色
android:textColorHint	设置提示文本颜色，默认为灰色
android:textSize	设置文字大小，推荐度量单位 sp
android:textStyle	设置字形，bold，italic，bolditalic 中的一种
android:typeface	设置文本字体，normal，sans，serif，monospace 中的一种
android:width	设置宽度

布局设计时，可以根据需要，在 XML 文件使用上面某些 EditText 的属性，来进行特殊的设置。例如，要求 EditText 中输入特定个数的字符，如身份证号、手机号码等，可以使用 android:maxLength="18"设定。下面给出一个例子，说明如何使用 EditText 的常用属性，见代码 3.24。

代码 3.24　edittext_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/edit_text1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="请输入用户名…"
```




```
        android:maxLength="40" />

    <EditText
        android:id="@+id/edit_text2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="请输入用户名…"
        android:maxLength="40"
        android:textColorHint="#238745" />

    <EditText
        android:id="@+id/edit_text3"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="请输入密码…"
        android:inputType="textPassword" />

    <EditText
        android:id="@+id/edit_text4"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="请输入电话号码…"
        android:inputType="phone" />

    <EditText
        android:id="@+id/edit_text5"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="请输入数字…"
        android:inputType="numberSigned" />

    <EditText
        android:id="@+id/edit_text7"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="请输入日期…"
        android:inputType="date" />

</LinearLayout>
```

编写 Java 程序代码，引用 edittext_layout. xml 定义的布局，运行应用就会看到 Edit 不同输入类型显示的效果(见图 3. 9 左图)。

在应用程序给出的界面上操作，可以体验 EditText 不同属性设置对输入的影响，了





图 3.9　Edit 不同输入类型和对 Edit 的不同操作

解如何使用这些属性来满足应用程序界面输入的需求。通过 EditText 的其他属性，还可以进一步修改提示文本和文本的字体、颜色和字形，可以设置 EditText 是否可编辑等。

在对应用界面操作过程中，注意在设置为 android:inputType="phone"的 edit_text4 中输入文本时，EditText 只接受电话号码输入的文本框，而且软键盘也变成拨号专用软键盘了。EditText 的 android:inputType 属性能够设置为"number" "numberSigned" "numberDecimal"等不同的值来控制输入的数字类型，分别对应 integer(正整数)、signed(带符号整数)和 decimal(浮点数)。也可以通过 android:inputType 来设置文本的类型，让输入法选择合适的软键盘，具体的值可以查 EditText 在 http://developer.android.com/reference/android 的文档。

在 android:inputType 属性设定输入的文本类型后，软键盘的转换是自动的。除了这个属性之外，EditText 的 android:imeOptions 属性也可以对软键盘进行控制，通过不同的值，在 Enter 键的位置显示出不同的按钮，例如"完成""搜索""去往"等。下面是 android:imeOptions 的几个常用的常量值，其值不同，软键盘显示的按钮也不同。

- actionUnspecified：Enter 按钮。
- actionGo：Go 按钮。
- actionSearch：Search 按钮。
- actionSend：Send 按钮。
- actionNext：Next 按钮。
- actionDone：Finish 按钮。

除了输入和对软键盘的控制之外，EditText 对输入后的文本操作也很灵活，Android 为 EditText 定义了很多处理方法，能够实现取值、全选、部分选择、获取选中文本。获取文本的操作在 Java 程序代码中实现，由事件处理器根据不同的操作，对文本进行不同的处理。下面通过一个例子来说明如何对 EditText 输入的文本进行取值、全选、部分选择

和获取选中文本,见图 3.9 右图。

界面的 XML 布局文件可以参考前面的内容,在纵向线性布局中设置界面显示的控件,其中 EditText 的属性 android:imeOptions 设置为"actionSearch",基于监听器的事件处理具体实现见代码 3.25。

代码 3.25　EditTextSearchActivity.java

```
public class EditTextSearchActivity extends Activity {
    private EditText editText;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.edittext_search);
        editText= (EditText) findViewById(R.id.edit_text);
        editText.setOnEditorActionListener(new OnEditorActionListener() {
            public boolean onEditorAction(TextView v, int actionId,
                    KeyEvent event) {
                switch(actionId) {
                case EditorInfo.IME_ACTION_SEARCH:
                    Toast.makeText(EditTextSearchActivity.this,
                            String.valueOf("开始搜索: ")+v.getText(),
                            Toast.LENGTH_SHORT).show();
                    break;
                }

                return true;
            }
        });
        //获取 EditText 文本
        Button getValue= (Button) findViewById(R.id.btn_get_value);
        getValue.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(EditTextSearchActivity.this,
                        editText.getText().toString(), Toast.LENGTH_SHORT)
                        .show();
            }
        });
        //让 EditText 全选
        Button all= (Button) findViewById(R.id.btn_all);
        all.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
```

```
                editText.selectAll();
            }
        });
        //从第 2 个字符开始选择 EditText 文本
        Button select=(Button) findViewById(R.id.btn_select);
        select.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Editable editable=editText.getText();
                Selection.setSelection(editable, 1, editable.length());
            }
        });
        //获取选中的文本
        Button getSelect=(Button) findViewById(R.id.btn_get_select);
        getSelect.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                int start=editText.getSelectionStart();
                int end=editText.getSelectionEnd();
                CharSequence selectText=editText.getText().subSequence
                        (start, end);
                Toast.makeText(EditTextSearchActivity.this, selectText,
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

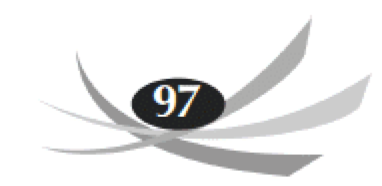
3.2.5 ImageView 控件

ImageView 控件是 Android 用于显示图片的控件,可以用于显示来自资源文件、Drawable 对象、Bitmap 对象或 ContentProvider 的 URI 等不同来源的图片,并能够通过各种属性来控制图片的各种显示选项,例如缩放和着色等。ImageView 的属性可以直接在 XML 布局文件中设置,也可以在 Java 应用程序中设置和修改。ImageView 是 View 的子类,具有 View 的属性,它在 XML 布局文件中的设置与 TextView 类似,见代码 3.26。

代码 3.26 在定义 XML 文件中 ImageView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
… />

    <ImageView
        android:id="@+id/image"
```




```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="center"
        android:src="@drawable/my_image" />
…
</LinearLayout>
```

在使用 ImageView 的过程中，一般会遇到两个问题：

一是在使用 ImageView 显示图片时，使用 android:src 属性设置图片的来源，默认状态下，运行时图形显示效果会有明显的边界，无法使图片充满整个 ImageView。解决方法是，把 android:src 属性设置图片源这条语句，改为 android:background="@drawable/my_image"。

另一个问题是如何通过 ImageView 的属性，来控制原始图片的尺寸、比例或者显示位置，以匹配 ImageView 本身设置的大小，运行时显示出设计预想的效果。解决方法是，使用 ImageView 最重要的一个属性 android:scaleType，根据需要设置对应的值。表 3.2 中列出了 ScaleType 的属性值和对应的含义。

表 3.2　ScaleType 属性值

属　性　值	含　　义
CENTER	图片居中显示，不执行缩放，图片大时会被裁减
CENTER_CROP	按照比例对图片进行缩放，充满 ImageView 控件，居中显示，截除图片多余部分
CENTER_INSIDE	图片比 ImageView 大，则根据比例对图片进行缩小并将其居中显示；图片比 ImageView 小，则不对图片进行处理，直接居中显示
FIT_CENTER	按照比例对图片进行缩放，并将图片居中显示
FIT_END	按照比例对图片进行缩放，将图片放置到右下角
FIT_START	按照比例对图片进行缩放，将图片放置到左下角
FIT_XY	拉伸或收缩图片，不保持原比例
MATRIX	从左上角开始绘制图片，超过 ImageView 的部分截掉

图 3.10 所示的 8 幅小图显示了比 ImageView 小的图片，在不同 ScaleType 属性的 ImageView 中显示的效果。这个例子中 Activity 中设计了一个 ImageView 和一个 Button，单击 Button 可以转换 ImageView 的 ScaleType 属性，并把这个属性值标注在 Button 上。

这个例子的 XML 布局文件可以参考前面的例子编写。其中原始图片是 160×240 像素，ImageView 的尺寸设置成 android:layout_width 的值为"match_parent"，android:layout_height 的值为"400dp"。ImageView 的单击事件通过 android:onClick 属性设置回调方法 changeScale()，使用 ImageView 的 setScaleType()方法实现其 ScaleType 的值，从而改变用户界面中图形的显示效果，见代码 3.27。

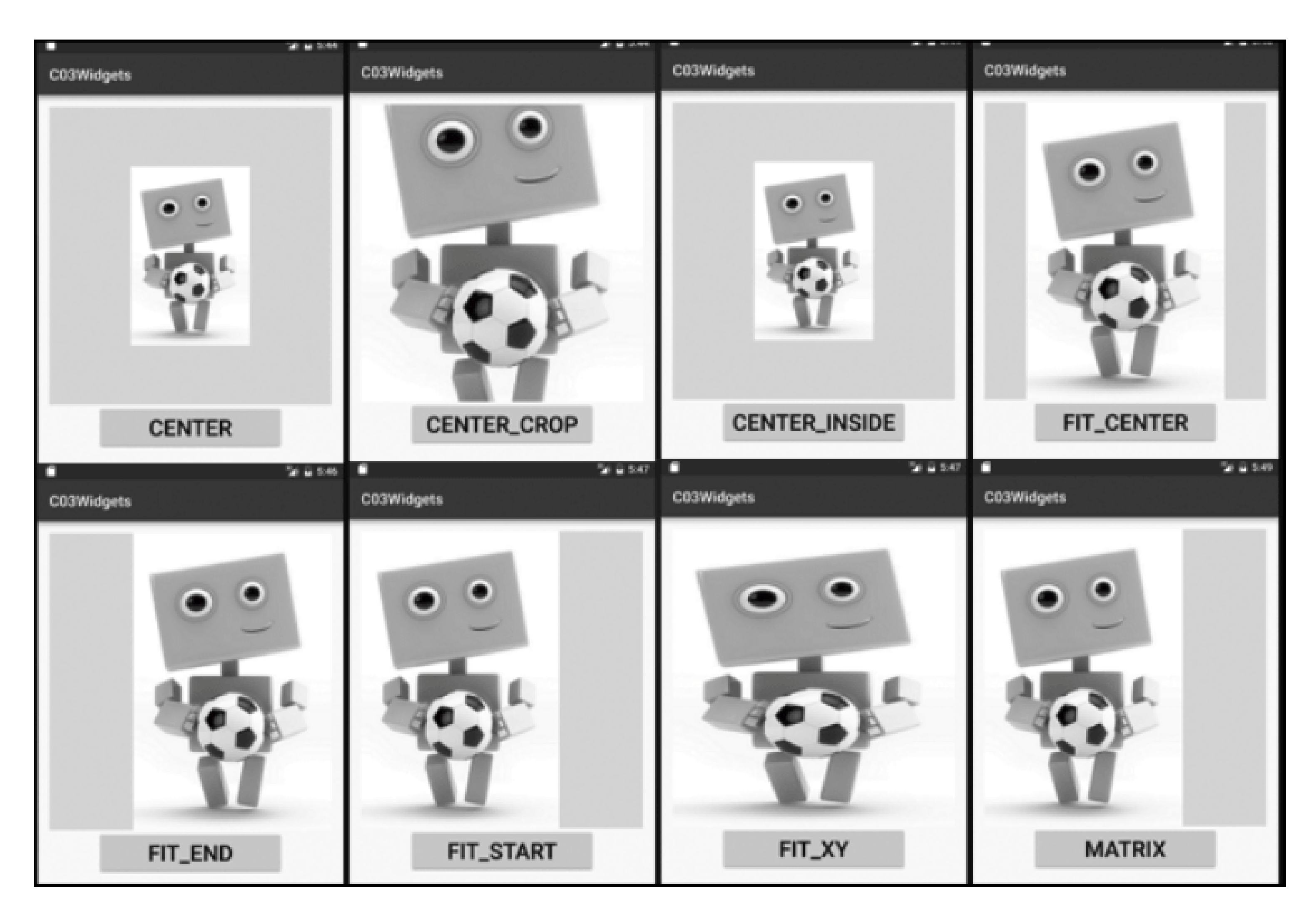


图 3.10　不同 ScaleType 属性的 ImageView 显示效果

代码 3.27　改变事件处理

```
public class ImageScaleActivity extends AppCompatActivity {
    Button btn;
    ImageView img;
    ImageView.ScaleType
    scale[]={ImageView.ScaleType.CENTER, …, ImageView.ScaleType.MATRIX};
    //属性值
    String scalestr[]={"CENTER", …,"MATRIX"};      //对应字符串
    int scaleIdx=0;

…

    public void changeScale(View view){
        scaleIdx++;
        scaleIdx=scaleIdx%scale.length;
        img.setScaleType(scale[scaleIdx]);
        btn.setText(scalestr[scaleIdx]);
    }
}
```

3.2.6　ProgressBar 控件

进度条是在 UI 进程中显示工作进程进度的一个重要工具。ProgressBar 是 Android 提供的一个进度条类型，表示运转的过程，例如发送短信、连接网络等，表示一个过程正在执行中。ProgressBar 的样式主要有普通圆形、大号圆形、小号圆形和标题型圆形等。

1. 普通圆形

对于进度条的样式一般只要在 XML 布局中定义就可以了。

```
<ProgressBar android:id="@+android:id/progress"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

此时,没有设置它的风格,那么它就是圆形的一直会旋转的进度条。

2. 大号圆形

如果给进度条设置一个 style 风格属性后,该 ProgressBar 就有了一个风格,这里大号 ProgressBar 的风格是:

```
style="?android:attr/progressBarStyleLarge"
```

3. 小号圆形

小号 ProgressBar 对应的风格是:

```
style="?android:attr/progressBarStyleSmall"
```

4. 标题型圆形

标题型 ProgressBar 对应的风格是:

```
style="?android:attr/progressBarStyleSmallTitle"
```

下面使用 ProgressBar 实现进度条显示的例子,见代码 3.28 和 3.29。

代码 3.28　ProgressBarActivity.java

```
public class ProgressBarActivity extends Activity {
    private static final int PROGRESS=0x1;
    private ProgressBar mProgress;
    private int mProgressStatus=0;
    private Handler mHandler=new Handler();

    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        setContentView(R.layout.progressbar_activity);
        mProgress=(ProgressBar) findViewById(R.id.progress_bar);

        //Start lengthy operation in a background thread
        new Thread(new Runnable() {
            public void run() {
                while (mProgressStatus<100) {
```

```
                    mProgressStatus=doWork();

                    //Update the progress bar
                    mHandler.post(new Runnable() {
                        public void run() {
                            mProgress.setProgress(mProgressStatus);
                        }
                    });
                }
            }
        }).start();
    }
}
```

代码 3.29　progressbar_activity.xml

```
<LinearLayout
    android:orientation="horizontal"
    …>
   <ProgressBar
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       style="@android:style/Widget.ProgressBar.Small"
       android:layout_marginRight="5dp" />
   <TextView
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="@string/loading" />
</LinearLayout>
```

3.3　界面效果处理

Android 的 API 中提供了一些特殊的界面效果处理方法，下面介绍这些方法。

3.3.1　文本处理

在 Android 中，有时候需要对文本进行各种特别的设置，如颜色、大小、首行缩进，或者是在一段文本中加入图片，甚至是书写一些特殊的公式。如果通过布局文件使用多个控件来实现，一方面使用起来特别复杂，增加了布局文件维护的难度，另一方面，如果加入了太多的控件，在页面加载时也要耗费更多的资源。在 HTML 中，可以使用各种标签来实现这些特殊效果，而在 Android 中有类似的机制，只不过不是使用标签来实现，而是使用 Spannable 对象来实现。

3.3.2　定义链接

在传统的 HTML 网页中，加一个<a>标记就可以让一段文字变成超链接的形式，可以单击到链接的地址。在 Android 界面中也能提供类似的功能。Android 界面的大多数文本一般通过 TextView 对象来显示，TextView 的 android:autoLink 属性设置可以实现这个功能。如果某个 TextView 的 android:autoLink 属性设置成"web"，则该 TextView 中网址形式的字符就会自动变成超链接的形式。下面使用一个简单的例子来说明设置的过程。

首先在/res/values/String.xml 中定义一个字符串，见代码 3.30。

代码 3.30　定义字符串资源

```
<string name="taobao_Mcommerce_android">移动电子商务开发实践: http://code.
google.com/p/taobao-sdk-for-android-platform/</string>
```

然后，在/res/layout/main.xml 布局文件中，把 TextView 的 android:autoLink 属性设置成"web"，见代码 3.31。

代码 3.31　设置 TextView 的链接属性

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

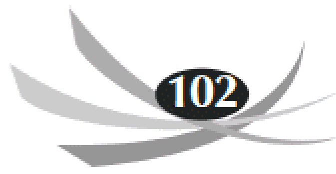
    <TextView
        android:id="@+id/textView"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:autoLink="web"
        android:text="@string/taobao_Mcommerce_android"
        android:textColor="#ff0000"
        android:textSize="24sp"
        android:textStyle="bold" />

</LinearLayout>
```

android:autoLink 的值还可以是"phone"或"email"，可以将字符串中的电话或邮件地址设置成超链接，如果将这个值设置为"all"，则对上述三种类型的字符串都设置成超链接。

3.3.3　文本样式

有时候，我们需要对文本显示的样式进行一些特殊的设置，例如突出显示 View 中部



分内容，可以使用 Android 的文本样式设置来完成。Android 的文本设置样式可以使用静态或动态方式设置内容的样式。

1. 静态方式设置文本样式

静态的方式是直接在/res/values/string.xml 中定义一个字符串变量，并指定其内容。如下所示：

```
<string name="styledText">在<b>TextView</b>中设置<i>静态</i>样式
</string>
```

字符串定义好后，就可以在 XML 布局文件或应用程序代码中引用这个字符串资源了。在字符串定义时，可以使用 XML 的标签。例如，＜i＞、＜b＞和＜u＞分别代表斜体、加粗和下划线，还可以使用＜sup＞、＜sub＞、＜strike＞、＜big＞、＜small＞和＜monospace＞等标签。这些标签不仅可以作用在 TextViews 上，而且对其他 View 也起作用。

使用 Activity 导入代码 3.32 中的布局文件，查看显示内容可知，在字符串中定义的格式效果完全显示出来。

代码 3.32 带样式的布局文件

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

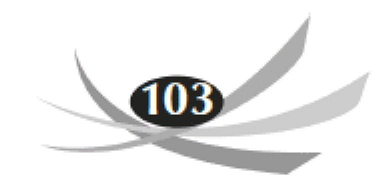
    <TextView
        android:id="@+id/tvStyled"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/styledText"
        android:typeface="monospace" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/italics"
        android:typeface="monospace" />

</LinearLayout>
```

2. 动态方式设置文本样式

动态方式设置文本样式，也就是通过应用程序代码编程的方式，来设置或改变 TextView 中的显示内容的格式。这种方式比静态方式要复杂，但是更具灵活性。



如果要动态设置文本样式，首先要将TextView中显示的内容设置为Spanable对象，代码如下：

```
tv.setText("字符串", TextView.BufferType.SPANNABLE);
```

通过上面的设置，TextView中的内容被存储成Spannable对象，然后就可以使用TextView的getText()方法获取Spannable对象了。具体获取Spannable对象的代码如下：

```
Spannable spn=(Spannable) tv.getText();
```

也可以先创建一个SpannableString对象，然后使用TextView的setText()方法将这个对象传递给TextView。代码如下：

```
SpannableString msp=new SpannableString("字符串");
tv.setText(msp);
```

获取或创建Spannable对象以后，就可以使用Spannable类提供的setSpan(Obj what,int start,int end,int flags)方法来进行样式的设置。setSpan()方法的参数说明如下：what是具体样式对象，所实现的类都在android.text.style包中；start则是该样式开始的位置；end对应的是样式结束的位置；参数flags定义在Spannable中的常量。常用的有：

- Spanned.SPAN_EXCLUSIVE_EXCLUSIVE，不包含两端start和end所在的端点，可表示为(a,b)。
- Spanned.SPAN_EXCLUSIVE_INCLUSIVE，不包含端start，但包含end所在的端点，可表示为(a,b]。
- Spanned.SPAN_INCLUSIVE_EXCLUSIVE，包含两端start，但不包含end所在的端点，可表示为[a,b)。
- Spanned.SPAN_INCLUSIVE_INCLUSIVE，包含两端start和end所在的端点，可表示为[a,b]。

例如，设置文本字符串的第2、3个字符显示为粗斜体的代码如下：

```
msp.setSpan(new StyleSpan(android.graphics.Typeface.BOLD_ITALIC), 0, 4,
Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
```

Spannable类与StyleSpan类具有类似作用，用来构建样式的其他类都在android.text.style包下，其中还包括：

- AbsoluteSizeSpan，是指绝对尺寸，通过指定绝对尺寸来改变文本的字体大小。
- BulletSpan，BulletSpan，着重样式，类似于HTML中的<li>标签的圆点效果。
- ForegroundColorSpan，字体颜色样式，用于改变字体颜色。

代码3.33中示例了使用动态方式设置一个字符串中不同字符不同样式的用法，显示效果见图3.11。

代码 3.33　设置文本外观

```
public class SpannableActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState)setContentView(R.layout.c03_
        spannable);
        TextView tv3=(TextView) this.findViewById(R.id.tv3);
        tv3.setText("默认、35 像素、默认一半、默认两倍\n"+
                "前景色、背景色\n"+"正常、粗体、斜体、粗斜体\n"+
                "下划线、删除线\n"+"下标、上标\n"+
                "电话、邮件、网址\n"+"项目符号",
                TextView.BufferType.SPANNABLE);
        Spannable spn=(Spannable) tv3.getText();
        //设置字体(default,default-bold,monospace,serif,sans-serif)
        spn.setSpan(new TypefaceSpan("monospace"), 0, spn.length(),
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
        //设置字体大小(绝对值,单位：像素)
        spn.setSpan(new AbsoluteSizeSpan(35), 3, 7,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
        spn.setSpan(new RelativeSizeSpan(0.5f), 8, 12,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //0.5f：默认字体的一半
        spn.setSpan(new RelativeSizeSpan(2.0f), 13, 17,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //2.0f：默认字体的两倍

        //设置字体前景色
        spn.setSpan(new ForegroundColorSpan(Color.MAGENTA), 18, 21,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //设置前景色为洋红色

        //设置字体背景色
        spn.setSpan(new BackgroundColorSpan(Color.CYAN), 22, 25,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //设置背景色为青色

        //设置字体样式正常,粗体,斜体,粗斜体
        spn.setSpan(new StyleSpan(android.graphics.Typeface.NORMAL), 26, 28,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //正常
        spn.setSpan(new StyleSpan(android.graphics.Typeface.BOLD), 29, 31,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //粗体
        spn.setSpan(new StyleSpan(android.graphics.Typeface.ITALIC), 32, 34,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //斜体
        spn.setSpan(new StyleSpan(android.graphics.Typeface.BOLD_ITALIC), 35,
                38, Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //粗斜体

        //设置下画线
```

```
        spn.setSpan(new UnderlineSpan(), 39, 42,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);

        //设置删除线
        spn.setSpan(new StrikethroughSpan(), 43, 46,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);

        //设置上下标
        spn.setSpan(new SubscriptSpan(), 47, 49,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //下标
        spn.setSpan(new SuperscriptSpan(), 50, 52,
                Spanned.SPAN_EXCLUSIVE_EXCLUSIVE); //上标
        //设置项目符号
        spn.setSpan(new BulletSpan(
                android.text.style.BulletSpan.STANDARD_GAP_WIDTH, Color.
                GREEN), 62, spn.length(), Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
    }
}
```

运行后显示如图 3.11 所示。



图 3.11　各种格式的文本

3.3.4　切换绘图

Android 的视图对象在应用过程中，存在不同的状态。例如按钮，具有单击按下、获得焦点或正常情况等状态。根据控件的状态，StateListDrawable 对象可以使用几种不同的图像来为控件指定背景图片，这称为切换绘图。实现切换绘图有两种方式：XML 文件定义和应用程序定义。

1. 使用 XML 文件定义状态选择列表

一般来说，首先要使用 XML 文件定义状态选择列表。这个文件需要创建在 res 目录下的 drawable 文件夹中。在一个 XML 状态列表文件中只有一个<selector>元素，其中包含多个<item>元素，用来定义状态及其对应的图片。代码 3.34 中示例了一个状态选择列表定义的语法。

代码 3.34　状态选择列表定义

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" /><!--pressed-->
    <item android:state_focused="true"
          android:drawable="@drawable/button_focused" /><!--focused-->
    <item android:state_hovered="true"
          android:drawable="@drawable/button_focused" /><!--hovered-->
    <item android:drawable="@drawable/button_normal" /><!--default-->
</selector>
```

<item>元素通过属性定义显示图片时的状态。代码 3.34 定义了四种状态，其中包括：

1）android:state_pressed

其值为布尔型，如果为"true"，当对象被按下（例如触摸/单击一个按钮）使用此选项；如果为"false"，当处于没有按下状态时，默认设置使用此选项。

2）android:state_focused

其值为布尔型，如果为"true"，当对象拥有输入焦点时应使用此选项（例如当用户选择一个文本输入）使用此选项；如果为"false"，当处于没有焦点状态时，默认设置使用此选项。

3）android:state_hovered

其值为布尔型，如果为"true"，当游标悬浮在对象之上时，使用此选项；如果为"false"，不是处于游标悬浮状态，默认使用此选项。通常，这种此选项使用的图片与焦点状态的选项相同。

4）默认状态

当控件的状态发生变化时，Android 会从第一个<item>元素开始查找，匹配当前状态的<item>元素被选中，这种选择不是基于最佳匹配的方式，只要求符合当前状态的最低标准。<item>元素使用 android:drawable 属性指定一个图片资源，这个属性是必要的。

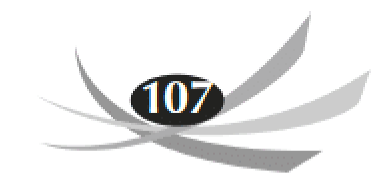
假设代码 3.34 定义的状态选择列表文件名为 button_bg.xml，在布局文件中可以直接引用代码 3.34 定义的资源，见代码 3.35。

代码 3.35　引用状态列表

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:background="@drawable/button_bg" />
```

2. 在应用程序中实现切换绘图

也可以使用编码的方式实现上面的功能，见代码 3.36。



代码 3.36　使用代码实现绘图切换

```
public class SetButtonBackground extends Activity {
    private ImageButton mButton;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Integer[] mButtonState={ R.drawable.android,
                R.drawable.android_focused, R.drawable.android_pressed };
        mButton=(ImageButton) findViewById(R.id.imgButton0);
        MyButton myButton=new MyButton(this);
        mButton.setBackground(myButton.setbg(mButtonState));
    }

    class MyButton extends View {

        public MyButton(Context context) {
            super(context);
        }

        public StateListDrawable setbg(Integer[] mImageIds) {
            StateListDrawable bg=new StateListDrawable();
            Drawable normal=this.getResources().getDrawable(mImageIds[0]);
            Drawable selected=this.getResources().getDrawable(mImageIds[1]);
            Drawable pressed=this.getResources().getDrawable(mImageIds[2]);
            bg.addState(View.PRESSED_ENABLED_STATE_SET, pressed);
            bg.addState(View.ENABLED_FOCUSED_STATE_SET, selected);
            bg.addState(View.ENABLED_STATE_SET, normal);
            bg.addState(View.FOCUSED_STATE_SET, selected);
            bg.addState(View.EMPTY_STATE_SET, normal);
            return bg;
        }
    }
}
```

除了 Button 之外，还可以类似的方式给其他的视图对象指定不同状态的背景。

3.3.5　叠加绘图

在用户界面显示图形或图像时，使用 LayerDrawable 对象可以将多个绘图资源按照顺序层叠起来，最后一个绘图资源将会被放在最上面，呈现出叠加视图的效果。实现叠加

视图，同样可以通过两种方式来实现：XML 文件和应用程序代码。

LayerDrawable 可以通过 XML 文件定义，这个 XML 文件需要保存在 res/drawable/文件夹中。其根元素为<layer-list>，其中包含多个<item>子元素，代码 3.37 是一个定义叠加绘图资源的 XML 文件例子。

代码 3.37　定义叠加绘图资源

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
      <bitmap android:src="@drawable/android_red"
        android:gravity="center" />
    </item>
    <item android:top="10dp" android:left="10dp">
      <bitmap android:src="@drawable/android_green"
        android:gravity="center" />
    </item>
    <item android:top="20dp" android:left="20dp">
      <bitmap android:src="@drawable/android_blue"
        android:gravity="center" />
    </item>
</layer-list>
```

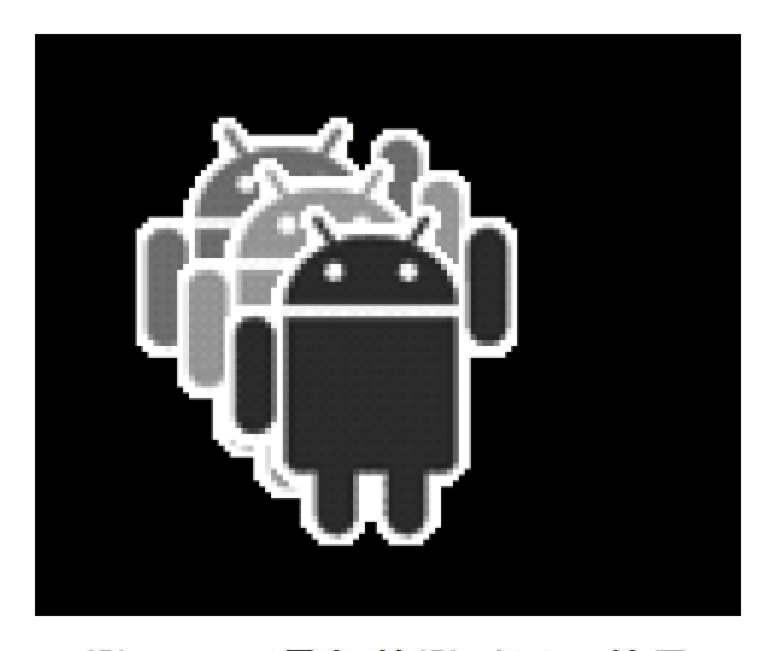


图 3.12　叠加的图形显示效果

代码 3.38 定义了三个绘图资源的叠加，最后定义的绘图资源显示在最上面。<item>元素可以定义绘图资源在四个方向的偏移量，其属性分别为 android：left、android：right、android：top 和 android：bottom。在默认情况下，所有的绘图资源都会发生缩放以适应包含其容器的尺寸。假设代码 3.38 文件名为 image_layers.xml，在布局文件中，可以在定义一个 ImageButton 时引用，见代码 3.39。显示效果见图 3.12。

代码 3.38　引用图形叠加资源

```
<ImageView
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:id="@+id/imgView"
    android:src="@drawable/image_layers" />
```

图 3.12 的效果也可以通过应用程序代码来实现。

代码 3.39　在应用程序中设置叠加绘图

```
BitmapDrawable d1=(BitmapDrawable) getResources().getDrawable(
        R.drawable.android);
d1.setGravity(Gravity.CENTER);
BitmapDrawable d2=(BitmapDrawable) getResources().getDrawable(
        R.drawable.android_focused);
d1.setGravity(Gravity.CENTER);
BitmapDrawable d3=(BitmapDrawable) getResources().getDrawable(
        R.drawable.android_pressed);
d3.setGravity(Gravity.CENTER);
Drawable drawableArray[]=new Drawable[] { d1, d2, d3 };
LayerDrawable layerDraw=new LayerDrawable(drawableArray);
layerDraw.setLayerInset(1, 10, 10, 0, 0);//set offset of 2 layer
layerDraw.setLayerInset(2, 20, 20, 0, 0);//set offset for third layer
ImageView imageView=(ImageView) findViewById(R.id.imgView);
imageView.setImageDrawable(layerDraw);
```

3.3.6　切换颜色

Android 的视图对象不仅可以根据状态定义切换的图形，还可以定义切换的颜色。根据控件的状态，Android 使用 ColorStateList 对象可以为控件定义几种不同的颜色。同样，定义切换的颜色可以有使用 XML 文件定义和程序代码实现两种方式。

切换颜色的状态列表也是通过 XML 文件定义的方式，与实现切换图片效果的方式类似，都是通过<selector>和<item>元素来定义的。不同之处是切换颜色必须定义<item>的 android:color 属性，其属性值为十六进制颜色，使用 RGB 值来指定，并且可选择 Alpha 通道。RGB 值始终使用＃字符开头，后面跟 Appha-Red-Green-Blue 信息，格式可以为＃RGB、＃ARGB、＃RRGGBB 和＃AARRGGBB。例如代码 3.40 定义了三种状态对应三种颜色。

代码 3.40　定义三种状态和颜色

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

   <item android:state_pressed="true" android:color="#ffff0000"/>
<!--pressed-->
   <item android:state_focused="true" android:color="#ff0000ff"/>
<!--focused-->
   <item android:color="#ff00ff00"/>
<!--default-->

</selector>
```

另一个与切换绘图设置的不同之处在于，XML 定义文件的位置位于 res/color/文件夹中。假设代码 3.41 的文件名为 edittext_color.xml，如果要在布局文件中的两个 EditText 控件引用这个资源文件，直接使用 android:textColor 属性，见代码 3.41。

代码 3.41　在布局文件中引用

```
<EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="text01"
        android:textColor="@color/edittext_color" />

<EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="text02"
        android:textColor="@color/edittext_color" />
```

创建一个例子 Activity，导入代码 3.41 中定义的布局资源文件，尝试分别单击这两个 EditText，查看改变焦点后文本颜色会发生什么变化。

3.4　本章小结

本章主要分为三个部分。第一部分介绍了 Fragment 的概念，Fragment 的生命周期以及使用 Fragment 静态创建和动态创建用户界面的具体步骤和代码。第二部分介绍了 Android 用户界面的事件处理机制和一些常用控件的用法。Android 用户界面的事件处理机制有两种，基于监听接口和基于回调机制。对于常用控件，使用简单的例子说明了按钮控件、Toast、文本控件、ImageView 和 ProgressBar 控件的使用，包括它们的布局、属性以及基于监听接口的事件处理机制。第三部分是界面效果处理的使用，说明了文本特殊格式、切换绘图、叠加绘图和切换颜色如何通 XML 文件和应用程序来实现。

第 4 章 菜单和动作条

设计应用程序的浏览模式是完善用户体验需要考虑的重要一环。如果浏览模式设计的不佳，用户无法快速定位所需的功能，或者出现错误定位使用户无法得到所需的功能，这些都会产生不好的用户体验。从 Android 3.0 开始，Android 系统的应用浏览模式发生了较大的改变，引入了向上和返回的设计原则，并且提供了相应的设计组件，其中包含菜单、动作条和浏览抽屉等。如果要实现准确的、一致的应用程序浏览，还需要理解浏览模式的原则，并且学会组件的使用。

4.1 菜单模式

在 Android 中，支持菜单视图元素的关键类是 android.view.Menu，每个 Activity 都会关联一个这种类型的菜单对象。一个菜单对象包含了一些菜单项和子菜单。菜单项由 android.view.MenuItem 类表示，子菜单由 android.view.SubMenu 类表示。菜单项具有的属性包括名称、菜单项 ID、分组 ID、顺序等。

菜单的定义与用户界面的其他可视控件类似，可以通过 XML 文件定义菜单资源，保存在 res 目录下的 menu 文件夹中，在 Java 程序中可以通过 ID 来获取定义的对象，进行操作。定义菜单资源的语法见代码 4.1 示例。

代码 4.1　定义菜单资源的语法

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@[+][package:]id/resource_name"
          android:title="string"
          android:titleCondensed="string"
          android:icon="@[package:]drawable/drawable_resource_name"
          android:onClick="method name"
          android:showAsAction=["ifRoom" | "never" | "withText" | "always" |
          "collapseActionView"]
          android:actionLayout="@[package:]layout/layout_resource_name"
          android:actionViewClass="class name"
          android:actionProviderClass="class name"
          android:alphabeticShortcut="string"
          android:numericShortcut="string"
          android:checkable=["true" | "false"]
```

```
        android:visible=["true" | "false"]
        android:enabled=["true" | "false"]
        android:menuCategory=["container" | "system" | "secondary" |
        "alternative"]
        android:orderInCategory="integer" />
    <group android:id="@[+][package:]id/resource name"
        android:checkableBehavior=["none" | "all" | "single"]
        android:visible=["true" | "false"]
        android:enabled=["true" | "false"]
        android:menuCategory=["container" | "system" | "secondary" |
        "alternative"]
          android:orderInCategory="integer">
      <item />
    </group>
    <item>
      <menu>
      ?<item />
      </menu>
    </item>
</menu>
```

4.1.1 菜单资源

对于所有类型的菜单，Android 提供了标准的 XML 格式来定义菜单项，所以除了在代码中实例化菜单之外，还可以在一个 XML 菜单资源中定义菜单和菜单项，然后在 Activity 中使用资源 ID 加载菜单资源。使用 XML 资源来定义菜单是一种推荐的方式。使用 XML 资源来定义菜单有许多优点，例如可以更好地体现菜单的结构，可以使逻辑代码和菜单内容分离，可以为不同的平台版本、不同的屏幕尺寸等提供可以替换的菜单配置。代码 4.1 给出了使用 XML 文件定义菜单资源的示例，下面针对 XML 文件定义菜单资源时使用的元素进行说明。

- <menu>。此元素用来定义菜单，用来包含菜单项。必须有一个<menu>元素作为菜单资源 XML 文件的根元素，其中可以包含一个或多个<item>和<group>元素
- <item>。此元素用来定义菜单项，每个<item>都表示一个菜单项，而且其还可以包含一个内嵌的<menu>元素，用来创建子菜单。
- <group>。此元素是可选的、不可见的，可以用来对菜单项进行分类，目的是使它们可以共享相同的属性，例如激活状态和可见性。

另外，经常使用下面几种<item>元素属性来定义菜单项的显示和行为：

- android:id。表示菜单项的唯一资源 ID，用来识别菜单项。
- android:icon。表示菜单项的显示图标，可以指定一个图片资源。
- android:title。表示菜单项的显示标题，这里指定一个字符串资源。

例如，代码4.2定义了一个简单的菜单资源XML文件game_menu.xml，其中使用了<menu>、<item>元素和<item>元素的一些属性。

代码4.2　简单的菜单资源文件

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:icon="@drawable/ic_new_game"
          android:title="@string/new_game"
          android:showAsAction="ifRoom"/>
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
          android:title="@string/help" />
</menu>
```

如果需要增加子菜单，需要在<item>元素中包含<menu>元素，子菜单可以起到将应用程序功能按照主题进行分类的作用。例如，在Microsoft Office套件应用的菜单条中都有"文件"、"编辑"等子菜单。

子菜单中菜单选项的定义与菜单类似，元素和属性的应用也相同，代码4.3给出了使用XML资源文件定义子菜单的简单例子。

代码4.3　使用XML文件定义子菜单

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file"
          android:title="@string/file">
        <!--"file" submenu-->
        <menu>
            <item android:id="@+id/create_new"
                  android:title="@string/create_new" />
            <item android:id="@+id/open"
                  android:title="@string/open" />
        </menu>
    </item>
</menu>
```

菜单项目在菜单列表的排列顺序由android:orderInCategory和android:menuCategory两个属性值之和确定的。数值之和越小排列越靠前，表示更重要。例如一个菜单项排列顺序为4和另一个菜单项排列顺序数为6，如果为列表形式排列的菜单，第一菜单项将出现在第二个菜单项的上面；如果为六项形式排列的菜单，第一菜单项将出现在第二个菜单项的左边。android:menuCategory被称为菜单类别属性，包含了四种预定义数值，分别为：

- container。表示菜单类别从0x10000开始，由常量Menu.CATEGORY_

CONTAINER 定义。

- system。表示菜单类别从 0x20000 开始，由常量 Menu. CATEGORY_SYSTEM 定义。
- secondary。表示菜单类别从 0x30000 开始，由常量 Menu. CATEGORY_SECONDARY 定义。
- alternative。表示菜单类别从 0x40000 开始，由常量 Menu. CATEGORY_ALTERNATIVE 定义。

如果要将定义好的菜单资源加载到 Activity 中，需要使用 MenuInflater. inflate()方法，在下面的章节中将介绍每种菜单的加载方式。

4.1.2 菜单类型

Android SDK 支持丰富的菜单类型，包括常规的菜单、子菜单、上下文菜单、图标菜单、二级菜单和替代菜单。此外，Android 3.0 推出了动作条，可以与菜单进行交互。Android 4.0 已经推出弹出式菜单，可以随时响应按钮单击或任何其他 UI 事件。

Android SDK 提供的菜单有三种基本类型：选项菜单、上下文菜单和弹出菜单。

选项菜单是一个 Activity 菜单项的主要集合，如果在这里加入操作，将会影响应用程序的全局。如果使用 Android 2.3 或者之前的 SDK 版本开发，用户可以使用菜单键打开选项菜单，但是对于 Android 3.0 或者更高的版本来说，选项菜单中的菜单项目是通过动作条与其他屏幕动作项目一起展现的。从 Android 3.0 开始，一些设备已经不支持菜单键，需要使用动作条来开发应用。

上下文菜单是当用户长按某个视图或视图元素后出现的浮动菜单，菜单中包含的动作是与用户所选择视图元素相关的。在 Android 3.0 和更高版本上进行开发，可以在选定的内容上使用上下文操作模式，显示相应的操作。这种模式在屏幕上方的操作条中显示影响所选的内容的操作项，并允许用户选择多个项目。

弹出菜单被固定在调用菜单的视图元素上，并且在一个垂直列表中显示菜单项目。在 Android 中，这些菜单都可以在 XML 资源文件中定义，并通过菜单资源文件中的 ID 加载到 Java 程序中。

下面分别介绍这三种类型的菜单。

1. 选项菜单

在 Android SDK 中，由于每个 Activity 都会关联一个菜单，因此不需要从头开始创建一个菜单对象。Android 的菜单创建，具体是由 Activity 的 onCreateOptionsMenu()回调方法来实现的，选项菜单的创建也可以由这个回调方法来实现。

选项菜单中包含的动作和选项与当前 Activity 上下文相关，并且根据 Android 系统版本的不同，其显示的位置也不同。如果使用 Android 2.3.x(API level 10)或者更低的版本，当用户按菜单键时，选项菜单的内容显示在屏幕的底部，可以最多显示 6 个带有图标按钮的无滚动条窗体；如果菜单项超过 6 个就需要使用扩展菜单项，这样这个窗体的最后一个按钮变成了 More，选中后会弹出一个包含多个菜单项的列表，可能还带有滚动条。图 4.1 中的示例显示了 Android 2.3.x 的选项菜单样式。

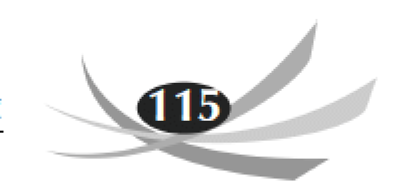




图 4.1　Android 2.3.x 选项菜单样式

如果使用 Android 3.0(API level 11)或者更高的版本,用户可以在动作条中使用选项菜单的菜单项。默认情况下,系统将所有的菜单项作为动作条的溢出操作,用户可以单击动作条最右边的溢出操作图标显示没有显示的菜单项;如果手机有菜单键,对于不在动作条中显示的菜单项,用户按菜单键则会看到剩余的菜单项。图 4.2 示例显示了 Android 3.0 的选项菜单样式。



图 4.2　Android 3.0 的选项菜单样式

选项菜单的菜单项既可以在 Activity 中声明,也可以在另一种更灵活的图形组件 Fragment 中声明。如果 Activity 和 Fragment 都为选项菜单声明了菜单项,而且合并在 UI 界面中,那么 Activity 中的菜单项优先显示,然后才按顺序把 Fragment 中菜单项添加到 Activity 中。如果要设定菜单项的顺序,也可以在<item>元素中添加 android:orderInCategory 的属性,重新按次序添加菜单项。

在 Activity 中,通过覆盖其 onCreateOptionMenu()方法来指定选项菜单,具体实现加载菜单;在 Fragment 中,也是覆盖其 onCreateOptionMenu()方法,通过菜单资源中定义的菜单 ID,获取菜单对象,赋值给声明的菜单类变量(见代码 4.4)。

代码 4.4　在 onCreateOptionsMenu()方法中获取菜单对象

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater=getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

在代码 4.4 中,getMenuInflater()方法返回了一个 MenuInflater 对象,用此对象来调用 inflate()方法,将菜单资源填充到菜单对象中。一旦菜单项被加载,onCreateOptionsMenu()

方法应该返回 true,使菜单可见。如果此方法返回 false,菜单是不可见的。对于 Android 2.3.x 和更低的版本来说,这个方法是在用户第一次打开菜单的时候由系统执行的;而对于 Android 3.0 和更高的版本来说,由于要在动作条中显示菜单项,系统在启动 Activity 时就调用此方法。

在 Android 应用程序中,也可以使用 Menu 类提供的 add()方法动态增加菜单项。Menu 类的 add()方法的参数说明如下:

- int groupId: 分组标识,其值相同的菜单项可以归为一组。
- int itemId: 菜单项 ID,代表菜单项的唯一编号,使用这个编号可以找到对应的菜单项。
- int order: 菜单项排列顺序(代表的是菜单项显示顺序,默认值是 0),其值越小表示越重要,优先显示。
- CharSequence title: String 类型的菜单项标题,表示需要在界面选项中显示的文字。除了使用字符串,还可以使用一个字符串资源,通过 R.java 文件常量文件。

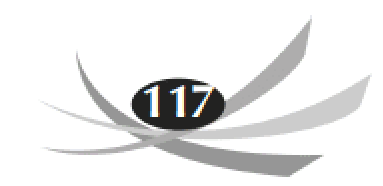
分组 ID、菜单项 ID 和排列属性都是可选的,如果不想特别指定的话可以使用 Menu.NONE。代码 4.5 给出了一个简单的例子,示例了如何使用 add()方法动态加载三个菜单项。

代码 4.5　使用 add()方法动态加载菜单

```
@Override
public boolean onCreateOptionsMenu(Menu menu){
    //call the base class to include system menus
    super.onCreateOptionsMenu(menu);
    menu.add(0 //Group
        ,1 //item id
        ,0 //order
        ,"append"); //title
    menu.add(0,2,1,"item2");
    menu.add(0,3,2,"clear");

    return true;
}
```

对于菜单选项的单击事件,Android 系统使用专门的方法来进行处理。当用户单击菜单项时,系统会调用 Activity 的 onOptionsItemSelected()方法,并且将用户单击的菜单项对象(MenuItem)传递给该方法。在这个方法中,可以用 getItemId()方法来获取菜单项的资源 ID,针对不同的菜单项,进行不同的操作。代码 4.6 中,通过 getItemId()获取菜单 ID 后,通过判断 ID 不同的值,实现在用户单击菜单项后,指定的文本框中显示出所单击的菜单选项标题。当然,这个例子为了简单,编写的事件响应操作代码都相同,但实际应用程序中,对应于每个菜单选项的事件处理,都对应了其响应的功能实现代码或方法。



代码 4.6　使用 onOptionsItemSelected()方法处理菜单项事件

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        TextView txt=(TextView)findViewById(R.id.txt);
        switch(item.getItemId()) {
            case 1:
                txt.setText("you clicked on item "+item.getTitle());
                return true;
            case 2:
                txt.setText("you clicked on item "+item.getTitle());
                return true;
            case 3:
                txt.setText("you clicked on item "+item.getTitle());
                return true;
        }
        return super.onOptionsItemSelected(item);
    }
```

如果被选的菜单项得到成功处理，则 onOptionsItemSelected()返回 true 值，否则，需要调用父类的 onOptionsItemSelected()方法继续处理。如果 Activity 中包含 Fragment，那么系统首先会调用 Activity 中 onOptionsItemSelected()方法，然后才是每个 Fragment 中的方法 onOptionsItemSelected()，直到有一个方法返回 true 值，否则所有的 onOptionsItemSelected()方法都会被调用了。

除了使用 onOptionsItemSelected()之外，还可以使用监听器来响应和处理事件，这种方式需要实现 OnMenuItemClickListner 接口以及其 onMenuItemClick()方法(见代码 4.7)。

代码 4.7　定义 OnMenuItemClickListner 监听器

```
public class MyResponse implements OnMenuClickListener {
    …
    @override
    boolean onMenuItemClick(MenuItem item) {
        //coding
        return true;
    }
}
```

与传统的 Java 事件处理程序类似，Android 应用程序在使用监听器处理事件时，也需要对监听器进行注册。对于代码 4.7 定义的监听器，可以使用下面的代码注册：

```
MyResponse myResponse=new MyResponse(…);
menuItem.setOnMenuItemClickListener(myResponse);
```

如果同时定义了 onOptionsItemSelected()方法和监听器处理方法，单击菜单项就会首先执行监听器中的 onMenuItemClick()方法。如果 onMenuItemClick()方法返回值为 true，则表示单击菜单项的事件处理已经完成，就不会执行 onOptionsItemSelected()方法；如果返回值为 false，才执行 onOptionsItemSelected()方法。另外，在菜单资源文件中 Android 系统还为菜单项提供了 android:onClick 属性，可以定义菜单项处理单击事件的方法。

如果多个 Activity 都拥有相同的菜单，可以定义一个只有 onCreateOptionsMenu()和 onOptionsItemSelected()方法的 Activity，在其中实现这个菜单，然后让其他类来继承该类。如果想在子类中添加新的菜单项，则只需重写 onCreateOptionsMenu()方法，并且调用 super.onCreateOptionsMenu()方法创建父类的菜单项，然后再使用 add()方法添加新的菜单项。

但是 onCreateOptionsMenu()方法是用来初始化菜单的状态，只能在菜单刚被创建时才会执行，所以不能用这个方法而在 Activity 的生命周期中修改菜单。如果要想动态改变选项菜单，就要实现 onPrepareOptionsMenu()方法，系统会将当前使用菜单对象传递给该方法，可以在这个方法中修改菜单。Android 2.3 或更低的版本中，系统会在每次菜单打开的时候调用一次 onPrepareOptionsMenu()方法；而在 Android 3.0 及以上版本，由于选项菜单是在动作条中显示的，此选项菜单总是打开的，因此必须调用 invalidateOptionsMenu()方法请求系统调用 onPrepareOptionsMenu()方法执行更新操作。

下面我们使用一个简单的例子，来说明在应用程序中如何实现选项菜单。从初始状态到完成菜单事件处理，分为七个步骤，下面进行具体介绍。

（1）新建一个 Activity。在 Android 项目中新建一个 Activity。

（2）创建资源文件夹。在 Android 项目的/res 目录下，使用 Eclipse 的 File→New→Folder 创建一个新文件夹，命名为 menu，作为菜单资源文件存储的目录。如果这个目录已经存在，就使用已存在的目录。

（3）创建菜单 XML 文件。在 menu 目录下，使用与创建布局文件类似的步骤，创建菜单 XML 文件，命名为 my_options_menu.xml，根节点元素为<menu>。文件的内容如下：

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
</menu>
```

（4）添加 Menu Items。使用<item>子元素，在 my_options_menu.xml 中添加所需的菜单选项。代码如下：

```
<item android:id="@+id/about"
android:title="About" />
<item android:id="@+id/help"
android:title="Help" />
```

（5）创建 Menu Items 的图标。如果菜单选项需要设定图标，可以把要使用的图标用

PNG文件格式存放在应用程序的drawable目录下。默认情况下Eclipse在drawable下创建了low、medium、和high三个子文件夹，图标文件可以放在任意文件夹中。

图标的指定使用item的icon属性，具体语法见下面代码：

```
<item android:id="@+id/about"
android:icon="@drawable/about"
android:title="About" />
<item android:id="@+id/help"
android:icon="@drawable/help"
android:title="Help" />
```

(6) 获取菜单资源。菜单资源文件定义完成后，要显示在用户界面上，需要Activity加载定义好的菜单，在其onCreateOptionsMenu()方法中通过菜单资源文件名，将定义好的菜单实例化，获取并加载这个菜单对象。下面的代码是典型的菜单资源获取方法。

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater=getMenuInflater();
    inflater.inflate(R.menu.my_options_menu, menu);
    return true;
}
```

(7) 响应Item选择事件。菜单加载后，需要完成的就是菜单选项的处理了。最简单的方法就是使用Activity提供的onOptionsItemSelected()方法。代码如下：

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.about:
            startActivity(new Intent(this, About.class));
            return true;
        case R.id.help:
            startActivity(new Intent(this, Help.class));
            return true;
        default:
            return super.onOptionsItemSelected(item);
        }
}
```

到此为止，菜单的建立就完成了。在菜单的具体实现过程中，并不是每一个步骤都需要完成，可以根据具体的情况省略。

2. 上下文菜单

在桌面系统的用户界面中，当用户使用鼠标右击界面视图元素时，桌面系统就会弹出与此视图元素相关的动作列表。这个功能非常方便，用户可以很容易地找到与视图元素相关的功能。提供这种功能的菜单，称为上下文菜单。

Android 系统也支持相同的设计模式，但由于用户交互的设备不同，操作时界面的响应有所不同。Android 的上下文菜单可以通过触摸屏操作调出。当用户按住触摸屏上的视图元素保持一段时间，就可以调出相关动作列表的上下文菜单。Android 系统可以为任何视图提供上下文菜单，但是通常在 ListView、GridView 中的项目上使用，或者在其他视图集合中的项目上使用。

Android 系统定义了两种模式的上下文菜单：

(1) 浮动模式。如果用户在视图元素上执行一个长单击(按住并保持)事件，上下文菜单项浮动列表会弹出，类似对话框，显示在原有视图的上面，覆盖原有的部分用户界面(见图 4.3 左图)。用户可以每次在浮动菜单中选择一个可执行的动作。

(2) 动作模式。这种模式是 ActionMode 的系统实现，可以在屏幕顶部显示上下文动作条动作条，其中的菜单项是影响所选视图元素的动作。当这种模式被激活，用户可以在使用上下文菜单的动作条中选择一个或多个动作。但是，这种模式只有在 Android 3.0 或者更高版本的可用，是使用上下文菜单的推荐模式。

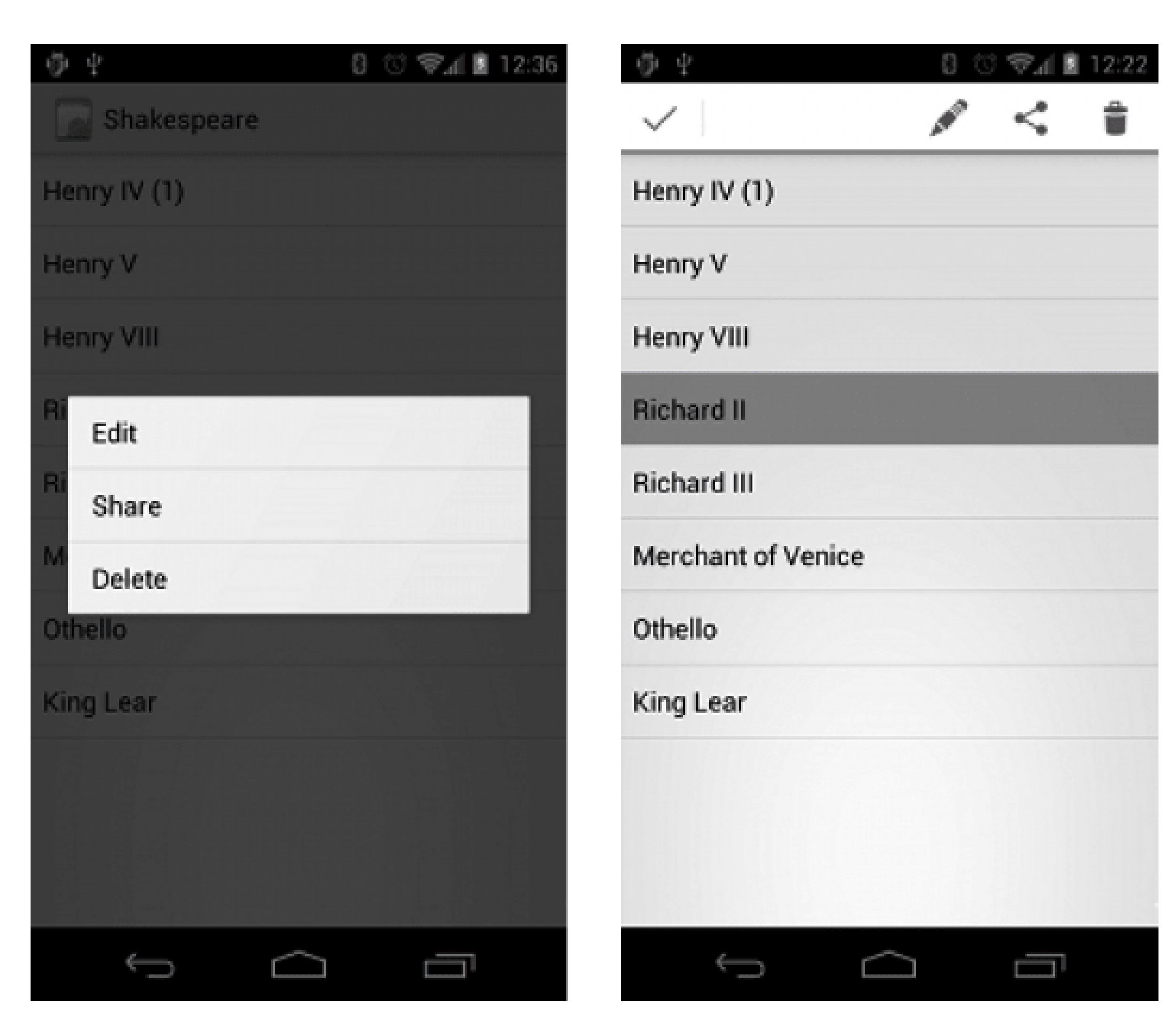


图 4.3 菜单模式

上下文菜单的加载与选项菜单类似，都是在 Activity 中通过特定方法中创建和加载，但具体在 onCreateContextMenu()方法中实现，而不是在 onCreateOptionsMenu()方法中实现。在实现 onCreateContextMenu()方法时，上下文菜单所依赖的视图元素通过参数指定。

上下文菜单的加载与选项菜单在加载时有所不同。onCreateOptionsMenu()方法在每个 Activity(或者 Fragment)启动时自动调用。由于不是界面上的所有视图元素都需要上下文视图元素，只有通过 registerForContextMenu(view)方法注册的视图元素，才有可能创建对应的上下文菜单，因此只有用户长单击某个视图元素后才需要执行 onCreateContextMenu()方法。

下面我们使用一个简单的例子，来说明一下在应用程序中如何实现上下文菜单。具

体创建步骤的前五步与选项菜单相同，从第六步开始有所不同。具体描述如下。

(1) 新建一个 Activity Class。

(2) 创建资源文件夹。

(3) 创建 Menu XML 文件。

(4) 添加 Menu Items。

(5) 创建 Menu Items 的图标。

(6) 注册视图元素。

通过调用 resisterForContextMenu() 方法为视图元素注册上下文菜单。如果 Activity 使用了 ListView 或者 GridView，且想要其中的每个项目都提供一个相同的上下文菜单，那么需要将 ListView 或者 GridView 对象传递到 registerForContextMenu()方法中。

(7) 获取菜单资源。

在 Activity(或者 Fragment)实现 onCreateContextMenu()方法。被注册视图元素接收到一个长单击事件，那么系统将会调用这个方法，在这里可以创建上下文菜单(代码 4.8)。

代码 4.8　覆盖 onCreateContextMenu()方法

```
@Override
public void onCreateContextMenu (ContextMenu menu, View v,
                                 ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater=getMenuInflater();
    inflater.inflate(R.menu.context_menu, menu);
}
```

MenuInflater 对象允许使用菜单资源填充上下文菜单对象。onCreateContextMenu()方法的参数包含了用户选择的视图元素 v 和提供关于被选项额外信息的 menuInfo 对象。如果需要为 Activtiy 中若干个视图元素提供不同的上下文菜单，需要使用这些参数来确定要填充的上下文菜单。

对于上下文菜单选项的单击事件，Android 系统使用 onContextItemSelected()方法来进行处理。当用户单击上下文菜单项时，系统会调用 Activity 的 onContextItemSelected()方法，并且将用户单击的菜单项对象(MenuItem)传递给该方法。

响应 Item 选择事件，覆盖 onContextItemSelected()方法。具体实现时，可以用 getItemId()方法来获取菜单项的资源 ID，针对不同的菜单项，编写不同的代码实现其功能(见代码 4.9)。

代码 4.9　使用 onContextItemSelected()方法处理菜单项事件

```
public boolean onContextItemSelected(MenuItem item) {
    AdapterContextMenuInfo info=(AdapterContextMenuInfo) item.getMenuInfo();
    switch(item.getItemId()) {
        case R.id.edit:
```

```
            editNote(info.id);
            return true;
        case R.id.delete:
            deleteNote(info.id);
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

如果被选的菜单项得到成功处理，则 onContextItemSelected()返回 true 值，否则，需要调用父类的 onContextItemSelected()方法继续处理。与选项菜单类似，如果 Activity 中包含 Fragment，那么 Activity 将首先执行自己的这个方法，如果返回值为 false，则通过调用 super. onContextItemSelected(item)方法，单击事件将会在每个 Fragment 中的 onContextItemSelected()方法中传递，按照 Fragment 被添加的顺序一个接着一个，直到返回 true 或者全部执行完为止。

3. 弹出菜单

弹出菜单是在 API 级别 11 和更高版本上才有效的。弹出菜单是一个在视图元素上弹出的模式菜单。如果这个视图元素下方有空间，那么弹出菜单将显示在视图元素的下方，否则会显示在上方。弹出菜单与上下文菜单不同，上下文菜单是对选择内容有影响的操作。

弹出菜单的功能包括：

- 为关联到特殊内容的动作提供一个溢出模式的菜单。例如 Gmail 的邮件头部(见图 4.4)。
- 提供一个命令的第二部分。例如一个标记为 Add 的按钮，使用不同 Add 选项可产生一个弹出菜单。
- 提供一个类似 Spinner 的下拉菜单。

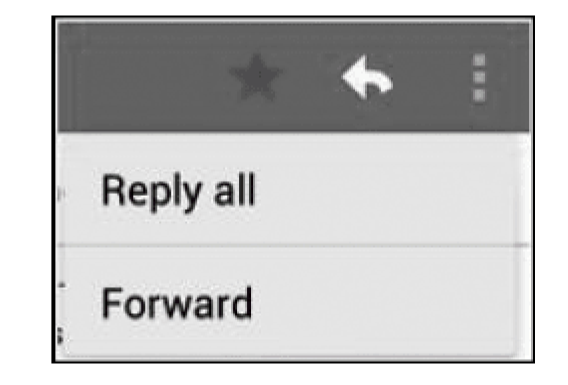


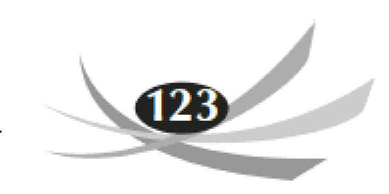
图 4.4 Gmail 的邮件头部

创建弹出式菜单的步骤与前两种菜单类似。具体创建过程中，前五步与前两种菜单相同，从第六步开始有所不同。

假设已经定义了菜单资源文件 popup_color_menu. xml(见代码 4.10)和用户界面布局文件 popmenudemo. xml(见代码 4.11)，则显示出弹出菜单还需要以下几步。

代码 4.10 弹出菜单资源文件定义

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_red" android:title="Red" />
    <item android:id="@+id/menu_green" android:title="Green"/>
    <item android:id="@+id/menu_blue" android:title="Blue"/>
</menu>
```



代码 4.11　用户界面布局文件

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
             android:orientation="vertical"
             android:layout_width="match_parent"
             android:layout_height="match_parent">

    <Button android:id="@+id/popupMenuBtn"
          android:layout_height="wrap_content"
          android:layout_width="wrap_content"
          android:text="Show me the Popup"
          android:layout_gravity="center"
          android:gravity="center"/>
</LinearLayout>
```

(1) 创建 PopupMenu 对象。

在所依附的视图元素的 Activity 初始化时,使用 PopupMenu 的构造方法实例化一个弹出菜单对象,并说明当前应用的 Context 和所依附的视图元素。

这里把实例化后的按钮和当前的 Activity 作为创建 PopupMenu 时的参数。

```
final Button btn= (Button) findViewById(R.id.popupMenuBtn);
final PopupMenu popupMenu=new PopupMenu(this, btn);
```

(2) 获取菜单资源,并导入到 popmenu 对象。

可以调用 PopupMenu. getMenu()来返回菜单对象。在 API 14 及高于 14 的,也可以用 PopupMenu. inflate()来导入。

```
popupMenu.inflate(R.menu.color_menu);
```

(3) 调用 PopupMenu. show()。

在按钮单击事件处理的 onClicklistener 中调用 PopupMenu. show()方法,显示预定义的弹出菜单。

```
btn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    popupMenu.show();
    }
});
```

(4) 菜单选项事件处理。

对弹出菜单选项的事件处理时,必须实现 PopupMenu. OnMenuItemClickListener 接

口并且通过调用 setOnMenuItemClickListener()方法把它注册给 PopupMenu 对象。当用户选择某个菜单选项时，系统会调用此接口中的 onMenuItemClick()回调方法进行处理。

在此例中，直接通过匿名内部类的方式，实现了 PopupMenu.OnMenuItemClickListener 接口，并在其 onMenuItemClick()方法中根据用户的选择改变背景的颜色，然后使其注册到按钮上。具体的代码实现和此例完整的程序见代码 4.12。

代码 4.12　创建弹出菜单

```
public class PopupMenuDemo extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.menudemo);

        final Button btn= (Button) findViewById(R.id.popupMenuBtn);
        final PopupMenu popupMenu=new PopupMenu(this, btn);

        popupMenu.inflate(R.menu.color_menu);

        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                popupMenu.show();
            }
        });

        popupMenu.setOnMenuItemClickListener(
                new PopupMenu.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {
                switch(item.getItemId()) {
                    case R.id.menu_red:
                        btn.setBackgroundResource(R.color.LightRed);
                        break;
                    case R.id.menu_blue:
                        btn.setBackgroundResource(R.color.DullBlue);
                        break;
                    case R.id.menu_green:
                        btn.setBackgroundResource(R.color.LightGreen);
                        break;
                }
```

```
                return true;
            }
        });

    }
}
```

4.1.3　菜单分组

菜单组是菜单项集合，可以用来为菜单项设置共同的属性。菜单组的设置可以使一组菜单选项的属性同时改变，呈现出共同的特性。例如：

- 使用 setGroupVisible()显示或隐藏组内所有选项。
- 使用 setGroupEnabled()启用或禁止组内所有选项。
- 使用 setGroupCheckable()说明组内所有的选项是否可选。

菜单组可以在菜单资源文件中定义，把<item>元素嵌套进<group>元素中来创建分组菜单；或者在 Android 应用程序中，使用带有分组 ID 的 add()方法创建分组。代码 4.13是一个在菜单资源文件中定义分组的简单例子。

代码 4.13　菜单分组定义

```
<?xml?version="1.0"?encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_save"
          android:icon="@drawable/menu_save"
          android:title="@string/menu_save" />
    <!--menu group-->
    <group android:id="@+id/group_delete">
        <item android:id="@+id/menu_archive"
              android:title="@string/menu_archive" />
        <item android:id="@+id/menu_delete"
              android:title="@string/menu_delete" />
    </group>
</menu>
```

在代码 4.13 中，分组菜单中的两个菜单项与第一个菜单项显示在同一个层次级别上，看上去没有什么区别。但是，能够使用 Android API 中的方法，通过引用分组 ID 同时修改其中两个菜单项的属性，而且系统不会将分组的菜单项给分开。

菜单组还可以用来显示应用程序中的选项开关，设置单选和多选两种方式，单选模式菜单见图 4.5。

但是，如果菜单组中的菜单选项是图标类型，则不能显示成复选框或单选按钮。如果选择了让图标菜单中的菜单项可复选，就必须在每次状态改变时通过手动更换图标与文本来指明复选的状态。

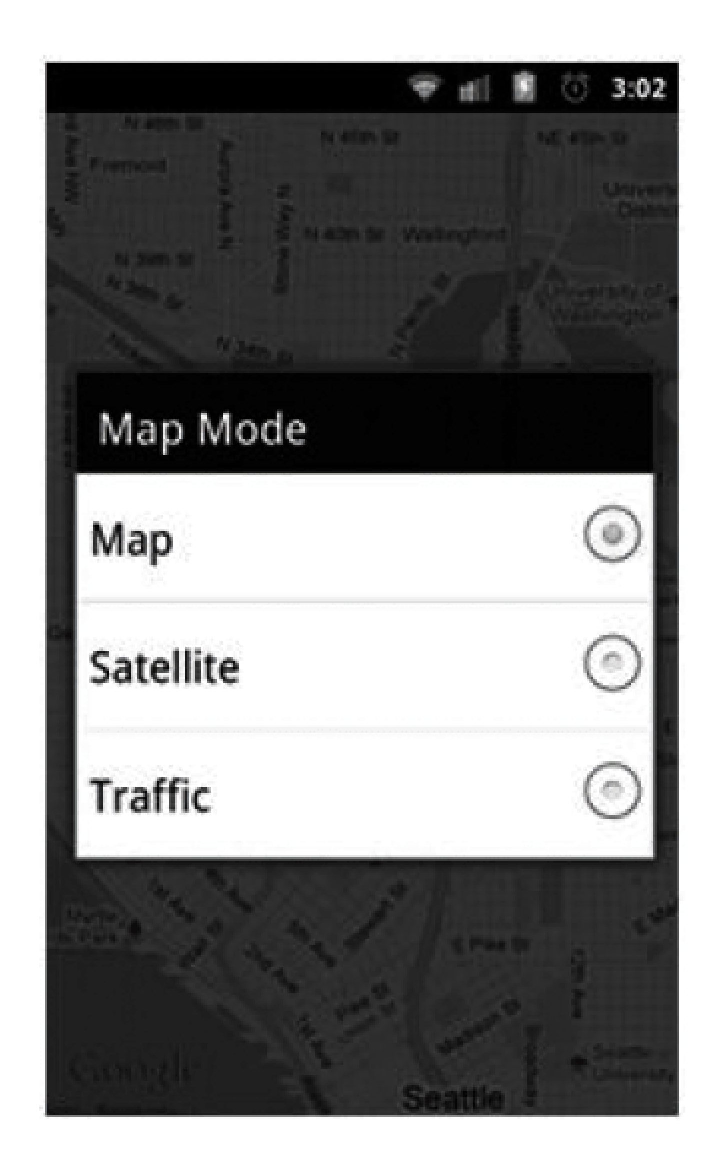


图 4.5　单选模式菜单

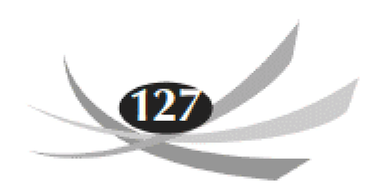
菜单资源文件中，<item>元素中的 android:checkable 属性用于给单独的菜单项定义是否可选，<group>元素中的 android:checkableBehavior 属性用于给一组菜单项定义可选类型（见代码 4.14）。

代码 4.14　为菜单项设置单选按钮

```
<?xml?version="1.0"?encoding="utf-8"?>
<menu?xmlns:android="http://schemas.android.com/apk/res/android">
    <group?android:checkableBehavior="single">
        <item?android:id="@+id/red"
            android:title="@string/red"?/>
        <item?android:id="@+id/blue"
            android:title="@string/blue"?/>
    </group>
</menu>
```

<group>元素的 android:checkableBehavior 属性可以有以下三种设置：single 代表菜单组中仅有一项能够被选（单选按钮）；all 代表所有菜单项都能够被选（复选框）；none 代表没有项目是可复选的。在<item>元素中可以使用 android:checked 属性给菜单项设置默认的选择状态，也可以用 setChecked()方法在代码中改变。

当一个可复选的菜单项被选择的时候，系统会调用对应被选择的菜单项的回调方法（如 onOptionsItemSelected()）。由于复选框或复选按钮不会自动地改变它们的状态，因此必须在这个方法中重新设置复选框的状态。一般使用 isChecked()方法来查询复选菜单的当前状态（被用户选择之前的状态），然后用 setChecked()方法设置选择状态（见代码 4.15）。



代码 4.15　在事件处理方法中设置菜单项的选择状态

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.vibrate:
        case R.id.dont_vibrate:
            if(item.isChecked()) item.setChecked(false);
            else item.setChecked(true);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

如果不用这种方式设置复选状态，那么当用户选择菜单项（复选框）的时候，它的可视状态将不会发生改变。

4.1.4　设置 Intent

菜单选项也可以创建 Intent 来启动另一个 Activity，这个 Activity 既可以是本应用程序中的，也可以是其他应用程序中的。如果确认了所需的 Intent 的特性以及初始化此 Intent 的菜单选项后，就可以在菜单选项事件响应的回调方法中使用 startActivity()运行此 Intent。

但是添加调用这个 Intent 对象的菜单项之后，如果不能确定用户设备上是否包含了处理这个 Intent 对象的应用程序，就有可能由于没有接收这个 Intent 对象的 Activity，导致这个菜单选项不会有任何作用，不能实现预期的功能，成为一个非功能性菜单选项。这个问题可以使用动态添加菜单项的方法来解决，Android 通过在设备上查找处理 Intent 对象的 Activity，动态地把菜单项添加到菜单中。

为了防止上述问题发生，可以在添加菜单选项具体采取一些措施，例如：

（1）使用分类 CATEGORY_ALTERNATIVE 和 CATEGORY_SELECTED_ALTERNATIVE 定义的 Intent。

（2）调用 Menu.addIntentOptions()方法，Android 系统会搜索能够接收这个 Intent 对象的应用程序，将菜单选项添加到菜单中。

（3）如果没有应用程序满足 Intent 的要求，就不添加菜单选项。

由于 CATEGORY_SELECTED_ALTERNATIVE 只用于处理当前屏幕上被选择的元素，因此只在用 onCreateContextMenu()方法创建菜单时使用这个分类（见代码 4.16）。

代码 4.16　动态添加 Intent 菜单选项

```
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);
```

```
    //Create an Intent that describes the requirements to fulfill, to
      be included
    //in our menu. The offering app must include a category value of Intent.
      CATEGORY_ALTERNATIVE.
    Intent intent=new Intent(null, dataUri);
    intent.addCategory(Intent.CATEGORY_ALTERNATIVE);

    //Search and populate the menu with acceptable offering applications.
    menu.addIntentOptions(
        R.id.intent_group,//Menu group to which new items will be added
        0,      //Unique item ID (none)
        0,      //Order for the items (none)
        this.getComponentName(),   //The current Activity name
        null,   //Specific items to place first (none)
        intent, //Intent created above that describes our requirements
        0,      //Additional flags to control items (none)
        null);  //Array of MenuItems that correlate to specific items (none)

    return true;
}
```

每找到一个 Intent Filter 与所定义的 Intent 对象相匹配的 Activity,Menu 类的 addIntentOptions()方法都会添加一个菜单项,这个菜单项使用其 Intent Filter 的 android:label 的属性值作为菜单项的标题,应用程序的图标作为菜单项的图标。addIntentOptions()方法返回被添加的菜单的个数。

4.2 动作条模式

动作条(ActionBar)是位于 Activity 顶端的一个图形控件,能够显示 Activity 的标题、图标、可能触发的动作、附加视图和其他交互控件,也可以用于在应用程序中导航。

动作条是用户浏览界面非常重要的设计元素,动作条主要提供以下功能:

- 支持应用程序内的一致导航和视图切换。
- 为极少使用的操作提供“溢出动作”模式,减少混乱。
- 提供专门位置来显示应用程序的标识,提示用户在当前应用程序中的位置。
- 突出重要操作,提供预访问模式。

对于大多数应用,动作条可以分割为应用图标、视图控制、动作按钮和溢出动作四个不同的功能区域,分别对应于图 4.6 中标识的“1”“2”“3”和“4”各区域。



图 4.6 ActionBar 的各区域

1. 应用图标

应用图标是应用程序的标识，需要的话可以使用不同的 logo 或标牌。如果当前不是应用的顶层界面，则在图标左边会有一个向左的箭头，表示“向上”按钮，使用户可以回到上一级界面(图 4.7 中右图的 logo 左边)。

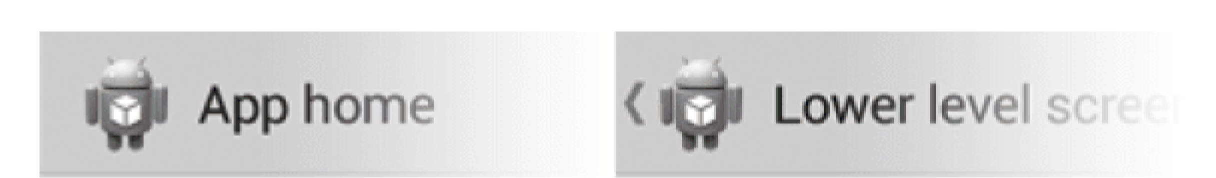


图 4.7 应用图标

2. 视图控制

如果应用程序可以在多个不同的视图中显示数据，动作条的“视图控制”能让用户在视图之间进行切换。例如，可以在下拉菜单和标签视图之间进行切换。如果应用程序不支持不同的视图，也可以使用这个区域显示非交互内容，例如应用程序标题或较长的标识信息。

3. 动作按钮

动作按钮用于显示应用程序中最重要的动作。Andriod 提供动作条的设计模式是为了方便用户选择与上下文环境相关的最重要的动作，那些直接显示在动作按钮区域上的图标或者文本被称为动作按钮。如果动作条的这部分区域不够容纳所有的动作，就自动移入后面的“溢出动作”。长单击图标就可以显示出“溢出动作”中隐藏的动作。

4. 溢出动作

平常很少用到的动作一般被放在这部分区域中。动作按钮区域和溢出动作区域中的动作按钮可能会根据屏幕的大小和形状发生变化。

Android 2.3 以及之前的系统使用设备上的返回键实现一个应用程序内部的浏览，这就是返回浏览模式。但是从 Android 3.0 版本开始，Android 设备使用虚拟浏览条，即动作条，代替了传统的物理按键。从 Android 3.0 (API level 11)开始，当 Activity 使用系统的默认主题模式显示时，动作条就出现在 Activity 窗口的顶部。当然也可以通过属性设定动作条的模式和增加动作条。

动作条的 API 首先在 Android 3.0 (API level 11)系统使用，但是要在 Android 2.1 (API level 7) 的版本中运行时，也可以导入相应的库兼容支持。在不同的 Android 版本中，大多数 API 都是相同的，只是所在的包不同。

如果支持 API level 11 低的版本，代码如下：

```
import android.support.v7.app.ActionBar
```

如果支持 API level 11 高的版本，代码如下：

```
import android.app.ActionBar
```

默认状态下，动作条跟在应用程序标题后面，显示在应用程序的顶部。如果 Activity 设置了弹出菜单，就可以从动作条以动作选项的方式直接访问。

使用动作条可以与许多控件结合使用，动作条支持的基本应用模式如下：

- 动作选项 Action Item。
- 动作视图 Action View。
- 动作提供器 Action Provider。
- 导航标签 Navigation Tab。
- 下拉菜单 Dropdown Menu。

4.2.1 添加 Action Item

从 Android 3.0(API Level 11)开始，动作条就被包含在使用 Theme.Hole 主题的 Activity 中，或者是这些 Activity 的子类中。由于动作条只能运行在 Android 3.0(API Level 11)或更高的版本上，动作条替代了选项菜单，并且替换了传统的应用程序标题栏。在添加 Action items 到动作条之前，需要打开 Manifest 文件，确认用户 SDK 的属性 targetSdkVersion 或 minSdkVersion 属性被设置为 11 或更大的数值。下面的代码是一个设置 targetSdkVersion 属性的例子。

```
<manifest …>
    <uses-sdk android:minSdkVersion="4"
              android:targetSdkVersion="11" />
    …
</manifest>
```

在这个例子中，应用程序运行要求的最小 API 版本是 4(Android 1.6)，但是目标 API 版本是 11(Android 3.0)。也就是说在 Android 1.6 版本支持下，应用程序可以完成基础重要的功能，但当应用程序运行在 Android 3.0 或更高的版本上时，系统就会给每个 Activity 应用全景主题，支持所有设计的界面和功能，包括动作条。如果要使用动作条 API 来进行添加导航模式和修改动作条样式的操作，就要把 minSdkVersion 属性设置为 11 或更大的值。

在查看 API 版本的同时，还需要确认下面列出的代码不存在，否则应用程序运行时动作条也不会显示。

```
<application
    android:theme="@android:style/Theme.NoTitleBar"
```

若在应用程序界面中不需要动作条，则把 Activity 的主题设置为 Theme. Holo. NoActionBar 就可以了。具体代码如下：

```
<activity android:theme="@android:style/Theme.Holo.NoActionBar">
```

动作条的显示和隐藏也可以在应用程序运行中动态调整。在 Activity 中使用 getActionBar()获取动作条对象后，使用 ActionBar 提供的 hide()方法和 show()方法可以直接改变动作条的显示状态。代码如下。

```
ActionBar actionBar=getActionBar();
actionBar.hide();
```

当动作条隐藏时，系统会调整 Activity 的显示大小，来填充当前有效的屏幕空间。在隐藏和删除动作条时，为了填充被动作条占用的空间，可能会导致 Activity 的重新布局。如果 Activity 有规律地隐藏和显示动作条，使用覆盖模式是较好的选择。设置动作条覆盖模式，需要给 Activity 创建一个主题，并且把 android: windowActionBarOverlay 属性设置为 true。

默认情况下，系统在动作条中使用应用的图标，这是通过<application>和<activity>元素中的 icon 属性指定。但是，如果设置了 logo 属性，动作条会使用 logo 属性指定的图片资源，而代替 icon 属性指定的资源。

由于 Android 3.0 及以上版本用动作条替代了选项菜单，当 Activity 首次启动时，系统会调用 onCreateOptionsMenu()方法创建 Action Item 类型的动作条，可以在这个方法中获取定义好的 XML 菜单资源。而且当 Action Item 的某个选项被选中时，对用户操作事件的获取和处理，Android 系统同样调用 onOptionsItemSelected()方法来进行处理。因此动作条中 Action Item 的定义与菜单选项相同，创建 Action Item 类型动作条和事件处理的过程也基本与选项菜单的步骤一致。

下面我们用一个简单的例子，来说明在应用程序中如何实现 Action Item 的动作条(具体代码见代码 4.17)。假设 Activity 的布局文件 main_activity_actions.xml 已经存在。

(1) 确认 Manifest 文件中的设置。

(2) 打开或创建一个 Activity Class。

(3) 若菜单资源目录/res/menu 不存在，创建这个目录。

(4) 创建 Menu XML 文件。

在 res/menu 目录下创建菜单资源文件 menu_actions.xml，在其中设置三个菜单选项。

代码 4.17　动作条 Action Item 的定义

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search"
        android:title="@string/action_search"/>
    <item android:id="@+id/action_compose"
        android:icon="@drawable/ic_action_save"
        android:title="@string/action_save" />
    <item android:id="@+id/itemHelp"
        android:icon="@drawable/ic_action_help"
        android:title="@string/btnHelp" />
</menu>
```

1. 设置显示动作选项

在 XML 文件中，能够通过设置<item>元素的 android:showAsAction="ifRoom"属性，使菜单项显示在动作条中。菜单项首先作为动作按钮显示在动作条中，如果没有足够的空间，其他菜单项会显示在溢出动作中。

<item>元素的 android:showAsAction 有一系列属性，下面列出重要的几个：

- never：此菜单项不显示在动作条中，当单击菜单按钮时，显示在列表中。
- ifRoom：如果动作条有足够空间，此菜单项显示在动作条上。
- always：此菜单项一直作为动作按钮显示，但不推荐使用，可能会引起视图重叠。
- withText：此菜单项使用定义的文本模式显示。

如果菜单项使用 android:title 和 android:icon 属性同时设置了标题和图标，那么默认情况下，动作条中动作项仅显示图标。如果要显示文本标题，就要给 android:showAsAction 属性添加 withText 设置，表示需要显示标题。代码如下：

```
android:showAsAction="ifRoom|withText" />
```

一般来说，即使不需要在动作条上显示文本标题，也应该在 XML 文件中给每个菜单项定义 android:title 属性，这是因为需要考虑下面几种情况：

- 如果动作条中没有足够的空间来显示动作项，菜单项就会显示在溢出动作中，在这里只能显示标题。
- 若只用图标来显示动作项，当用户长单击动作按钮时，可以显示带有标题的提示信息。
- 屏幕阅读器可以给有视觉障碍的用户朗读菜单项标题。

添加动作选项显示设置代码后的菜单资源文件见代码 4.18。

代码 4.18　设置显示动作选项

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search"
        android:showAsAction="ifRoom|withText" />
        android:title="@string/action_search"/>
    <item android:id="@+id/action_compose"
        android:icon="@drawable/ic_action_save"
        android:showAsAction="ifRoom|withText" />
        android:title="@string/action_save" />
    <item android:id="@+id/itemHelp"
        android:icon="@drawable/ic_action_help"
        android:showAsAction="ifRoom|withText" />
        android:title="@string/btnHelp" />
</menu>
```

2. 获取 Menu XML 定义的动作选项资源

在 Activity 中，动作选项资源的获取方式与选项菜单相同，都可以从预定义的菜单资

源文件中加载，在 onCreateOptionsMenu()方法中实现创建动作选项。这里，使用 MenuInflater 类的 inflate()方法，从 menu_actions.xml(见代码 4.18)文件中获取预定义的菜单项资源作为动作选项(见代码 4.19)。

代码 4.19 获取菜单资源

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //Inflate the menu items for use in the action bar
    MenuInflater inflater=getMenuInflater();
    inflater.inflate(R.menu.menu_actions, menu);
    return super.onCreateOptionsMenu(menu);
}
```

3. 响应 Action Item 选择事件

动作选项加载后，需要完成的就是动作选项的事件处理。与选项菜单相同，最简单的方法就是使用 Activity 提供的 onOptionsItemSelected()方法。当用户选择了一个动作时，系统会调用 Activity 的 onOptionsItemSelected()方法，并且传递 MenuItem 对象给这个方法。可以使用 getItemId()方法得到 android:id 属性为菜单项定义的 ID，来识别用户单击的动作(见代码 4.20)。

代码 4.20 Action Item 事件处理

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    //Handle presses on the action bar items
    switch(item.getItemId()) {
        case R.id.action_search:
            Toast.makeText(MainActivity.this, "Search is Selected", Toast.
            LENGTH_SHORT).show();
            return true;
        case R.id.action_save:
            Toast.makeText(MainActivity.this, "Save is Selected", Toast.
            LENGTH_SHORT).show();
              return true;
        case R.id.action_help:
              Toast.makeText(MainActivity.this, "Help is Selected", Toast.
              LENGTH_SHORT).show();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

如果在 Fragment 中添加菜单项，那么通过 Fragment 类的 onCreateOptionsMenu 回

调方法，当用户选择其中一个 Fragment 的菜单项时，系统会调用那个 Fragment 对象对应的 onOptionsItemSelected()方法。

4.2.2 添加 Actiion View

动作视图 Actiion View 是作为动作按钮的替代品显示在动作条中的一个可视构件。动作视图在不改变 Activity 或 Fragment 的情况下，可以给用户提供快捷的访问和丰富的操作。例如，如果有一个用于搜索的可选菜单项，可以用 SearchView 类来替代动作条上的搜索按钮。图 4.8 显示了从动作按钮展开为动作视图的例子。

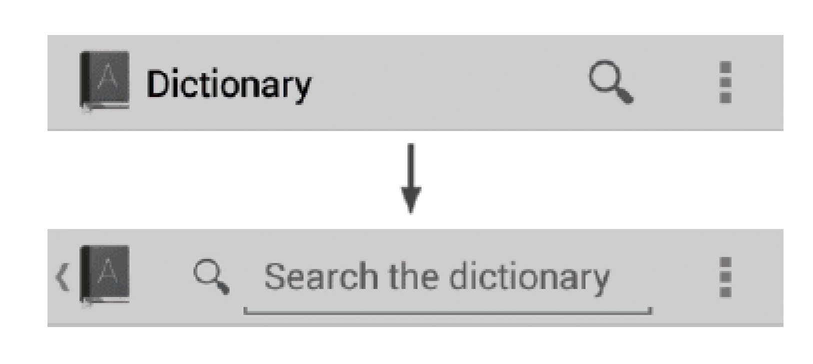


图 4.8 动作按钮操作

在菜单资源文件中，可以使用 <item> 的 android:actionLayout 或 android:actionViewClass 属性来指定一个布局资源或控件类，从而定义一个 Action View。例如，代码 4.21 中指定了一个 SearchView 控件作为 Action View。

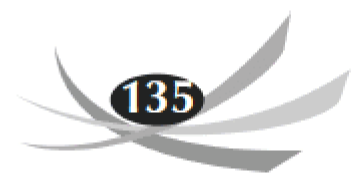
代码 4.21 在菜单资源中指定 Action View

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
     <item android:id="@+id/menu_search"
         android:title="@string/menu_search"
         android:icon="@drawable/ic_menu_search"
         android:showAsAction="ifRoom|collapseActionView"
         android:actionViewClass="android.widget.SearchView" />
</menu>
```

android:showAsAction 属性也可包含 collapseActionView 属性值，这个值是可选的。如果 android:showAsAction 属性得值包含了 collapseActionView，说明所声明的动作视图会折叠到一个按钮中，当用户选择这个按钮时再展开。否则，默认状态下这个动作视图是可见的，要占据动作条的有效空间。

如果为某个动作选项设定了动作视图，由于在用户选择这个动作选项时，系统会展开对应的动作视图，因此不必在 onOptionsItemSelected()回调方法中响应这个动作选项，进行相应的事件处理，而是直接交给动作视图的事件处理代码来完成。但是，如果 onOptionsItemSelected()中设置返回 true 值，系统还是会回调这个方法，动作视图则不会打开。当用户选择了动作条中的“向上”图标或按下了回退按钮时，系统也会把动作视图折叠起来。如果需要，可以在代码中通过 expandActionView()和 collapseActionView()方法来展开或折叠动作视图。

动作视图的事件处理代码放在 Activity 的 onCreateOptionsMenu()方法内，当事件发生时系统回调此方法，由注册的监听器捕获相应的事件，执行事件处理代码，实现选择动作视图后对应的功能。在 onCreateOptionsMenu()方法中，通过调用带有菜单选项 ID 的 findItem()方法来获取菜单选项，然后再调用 getActionView()方法获得动作视图中的



元素(见代码 4.22)。

代码 4.22　获得动作视图中的元素

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_activity_actions, menu);
    MenuItem searchItem=menu.findItem(R.id.action_search);
    SearchView searchView=(SearchView) MenuItemCompat.getActionView
    (searchItem);
    //Configure the search info and add any event listeners
    …
    return super.onCreateOptionsMenu(menu);
}
```

在对动作视图操作时,会有两种情况:展开和折叠。针对这两种操作,Android 提供了 OnActionExpandListener 接口的 onMenuItemActionExpand()方法和 onMenuItemActionCollapse()方法,在事件发生时进行回调。编写事件处理代码时,可以首先实现 OnActionExpandListener 接口,根据不同的操作,分别在对应的方法中实现相应的功能。然后创建一个 OnActionExpandListener 对象,即一个事件,并使用 setOnActionExpandListener()方法来注册这个事件。完成这些工作,系统就能够在动作视图展开和折叠时使用相应的回调方法进行处理了(见代码 4.23)。

代码 4.23　动作视图的事件处理

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);
    MenuItem menuItem=menu.findItem(R.id.actionItem);
    …
    //When using the support library, the setOnActionExpandListener()
      method is
    //static and accepts the MenuItem object as an argument
    MenuItemCompat.setOnActionExpandListener(menuItem,
    new OnActionExpandListener() {
        @Override
        public boolean onMenuItemActionCollapse(MenuItem item) {
            //Do something when collapsed
            return true;  //Return true to collapse action view
        }

        @Override
        public boolean onMenuItemActionExpand(MenuItem item) {
            //Do something when expanded
```

```
                return true;  //Return true to expand action view
            }
        });
    }
```

4.2.3 添加 Action Provider

与动作视图类似，动作提供器 Action Provider 使用一个定制的布局代替一个动作按钮。与动作视图不同，当单击动作提供器时，其可以显示子菜单，并需对所有选项进行控制处理。

有两种方式来使用 Action Provider：

- 在应用程序代码中，调用 MenuItem 类提供的方法 setActionProvider(ActionProvider)，直接在菜单选项上设置动作提供器。
- 在菜单资源文件中声明动作提供器。

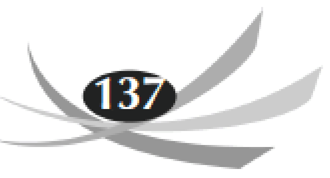
Android 系统提供了一些预定义的 Action Provider，例如，ShareActionProvider 提供了一个子菜单，菜单中包括 Google+、Hangouts 和 Messaging 等菜单选项，已经实现了其功能。通过使用这个预定义的动作提供器，可以将这些可能共享的应用列表直接在动作条上显示出来，供用户使用。使用如果要声明一个动作提供器，需在菜单资源文件中设定所对应的<item>元素的 android:actionProviderClass 属性，使用所选的预定义 Action Provider 的完整类名来指定动作提供器（见代码 4.24）。如果需要创建自己定义的动作提供器，也可以通过继承 Action Provider 类来定义。

代码 4.24　声明 Action Provider

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_share"
        android:title="@string/share"
        android:showAsAction="ifRoom"
        android:actionProviderClass="android.widget.ShareActionProvider" />
    ...
</menu>
```

由于每一个 Action Provider 类定义自身的动作行为，就不必在 onOptionsItemSelected() 监听动作事件了。如果需要在 onOptionsItemSelected() 方法中对其他的动作的事件进行处理，必须确认此方法的返回值为 false，从而保证动作提供器上菜单选项的事件发生时系统能够回调 onPerformDefaultAction() 方法，执行其事件处理代码。

前面的知识中提到过，在用户界面上选择选项菜单的任何一个选项，系统都会回调 onOptionsItemSelected() 方法，而动作条是 Android 高级版本中选项菜单的替代控件，所以一般情况下，可以把动作提供器的事件处理也放在 onOptionsItemSelected() 方法中，系统会在选择动作提供器时回调这个方法。



但值得注意的是，如果动作提供器定义了一个子菜单，则在用户打开子菜单列表或选择子菜单选项时，Activity 无法接收 onOptionsItemSelected()的回调，事件处理代码必须写在 onPerformDefaultAction()方法中。

4.2.4　使用系统定义的 Action Provider

Android 系统定义了一些 Action Provider，提供了丰富的功能。每个 Action Provider 类都定义了自己布局（例如子菜单）和动作行为，其他的应用程序在使用这些 Action Provider 时，可以直接使用其功能，由 onPerformDefaultAction()回调方法中系统定义的代码来处理事件，不需要再在 onOptionsItemSelected()方法中监听动作和进行事件处理。但是，尽管预定义的 Action Provider 提供了其在溢出菜单中所能执行的默认操作，应用程序的 Activity（或 Fragment）也能够通过处理来自 onOptionsItemSelected()回调方法的单击事件，来重写这个默认操作。

下面，我们把系统预定义 ActionProvider 之一的 ShareActionProvider 类作为例子，说明如何使用这些 Action Provider。

ShareActionProvider 类提供"共享"动作，它负责创建子菜单的所有逻辑，包括共享视图的封装、单击事件的处理以及溢出菜单中的选项显示等。在使用这个类时，所需要编写的唯一的代码就是给对应的菜单项声明动作提供器，并指定共享的 Intent 对象。如果要在动作条中提供一个"共享"动作，来充分利用安装在设备上的其他应用程序，例如把一张图片共享给消息或社交应用程序使用，则使用 ShareActionProvider 类是一个有效的方法，见图 4.9。

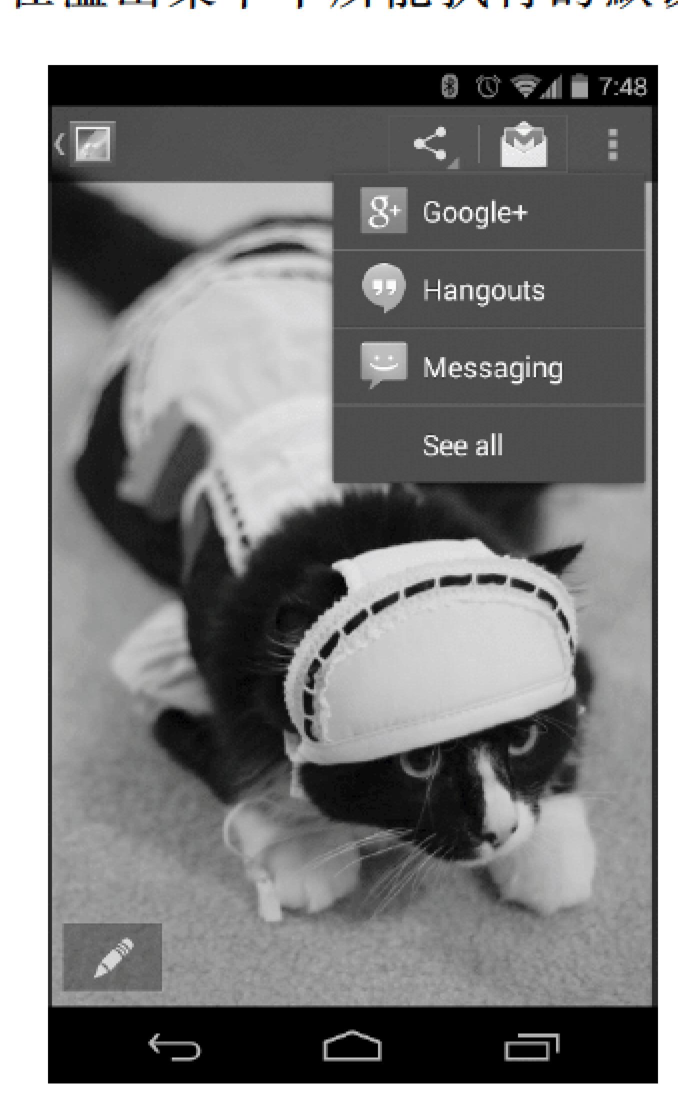


图 4.9　ShareActionProvider 的使用

要使用 ShareActionProvider 实现共享动作，需要再完成下面几个工作：

(1) 打开或创建 Activity Class。

(2) 定义菜单资源文件。

定义 Action Bar 上的动作选项，并在<item>元素下，定义 actionProviderClass 属性的值为 ShareActionProvider 的完整类名（见代码 4.24）。

(3)获取 Action Provider 对象。

在 Activity 中定义一个 ShareActionProvider 类型的私有变量 mShareActionProvider，然后在 onCreateOptionsMenu()方法中，使用 getActionProvider()方法获取跟 menu_share 菜单项匹配的 ShareActionProvider 对象，调用 setShareIntent()方法是定义要用于共享的 Intent 对象。

代码 4.25 中定义了 getDefaultIntent()方法初始化 Action Provider，但是当在 Intent 中实际使用的内容设定或改变时，必须调用 mShareActionProvider.setShareIntent()更

新共享 Intent。

代码 4.25　使用 ShareActionProvider

```
private ShareActionProvider mShareActionProvider;

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_activity_actions, menu);

    //Set up ShareActionProvider's default share intent
    MenuItem shareItem=menu.findItem(R.id.menu_share);
    mShareActionProvider=(ShareActionProvider)
            MenuItemCompat.getActionProvider(shareItem);
    mShareActionProvider.setShareIntent(getDefaultIntent());

    return super.onCreateOptionsMenu(menu);
}

private Intent getDefaultIntent() {
    Intent intent=new Intent(Intent.ACTION_SEND);
    intent.setType("image/*");
    return intent;
}
```

代码 4.25 中 ShareActionProvider 对象处理所有的跟这个菜单项有关的用户交互，并且不需要处理来自 onOptionsItemSelected()回调方法的单击事件。如果需要重新定义菜单项的功能，可以重写 onOptionsItemSelected()中对应的处理代码。代码 4.26 中在 onOptionsItemSelected()回调方法中重写了 menu_share 菜单项的单击事件处理代码，因此当用户单击这个菜单项时，系统不再执行原有的菜单功能，而是实现重写后的代码。

代码 4.26　重写 onOptionsItemSelected()中对应的处理代码

```
@Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);

        //Get the ActionProvider for later usage
        provider=(ShareActionProvider) menu.findItem(R.id.menu_share)
            .getActionProvider();
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
```

```
        switch(item.getItemId()) {
        case R.id.menu_share:
            doShare();
            break;
        default:
            break;
        }
        return true;
    }

    public void doShare() {
        //populate the share intent with data
        Intent intent=new Intent(Intent.ACTION_SEND);
        intent.setType("text/plain");
        intent.putExtra(Intent.EXTRA_TEXT, "This is a message for you");
        provider.setShareIntent(intent);
    }
```

默认情况下,ShareActionProvider 对象会基于用户的使用频率来保留共享目标的排列顺序。使用频率高的目标应用程序会显示在下拉列表的上面,并且最常用的目标会作为默认共享目标直接显示在动作条。默认情况下,排序信息被保存在由 DEFAULT_SHARE_HISTORY_FILE_NAME 指定名称的私有文件中。如果只使用一种操作类型 ShareActionProvider 类或它的一个子类,可以继续使用这个默认的历史文件;但是,如果使用了不同类型的多个操作的 ShareActionProvider 类或它的子类,则需要调用 setShareHistoryFileName()方法,为每种 ShareActionProvider 类都指定自己独立的 XML 历史文件。

4.2.5 自定义动作提供器

要创建自己的动作提供器,只需继承类,并且实现合适的回调方法。下面是需要实现的几个重要回调方法:

- ActionProvider()。构造方法负责传递应用程序的 Context 对象,这个对象需要保存在一个成员变量中,以便其他的回调方法使用。
- OnCreateActionView()。在这个方法中,可以给菜单项定义动作视图。使用从构造方法中接收的 Context 对象,实例化一个 LayoutInflater 对象,并且用 XML 资源来填充动作视图,然后注册事件监听器(见代码 4.27)。

代码 4.27 创建自己的 ActionProvider 动作视图

```
public View onCreateActionView() {
    //Inflate the action view to be shown on the action bar.
    LayoutInflater layoutInflater=LayoutInflater.from(mContext);
```

```
    View view=layoutInflater.inflate(R.layout.action_provider, null);
    ImageButton button=(ImageButton) view.findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            //Do something…
        }
    });
    return view;
}
```

- onPerformDefaultAction()。系统会在选中溢出菜单中的菜单选项时，调用这个方法，并且动作提供器执行这个选中菜单项的默认操作。如果是自定义的动作提供器，使用 onPrepareSubMenu()回调方法创建了子菜单，即使把这个动作提供器放在溢出菜单中，子菜单也会显示。因此，当自定义动作提供器中子菜单存在时，系统不会回调 onPerformDefaultAction()方法。

4.2.6 添加 Navigation Tab

对于应用中不同视图之间的切换和展现，动作条上的导航标签是一个很好的工具，而且导航标签适用于在不同的屏幕尺寸上显示。例如当屏幕足够宽，导航标签就显示在动作按钮旁的动作条上；而屏幕较窄时，导航标签就显示在分离的横条中(见图 4.10)。

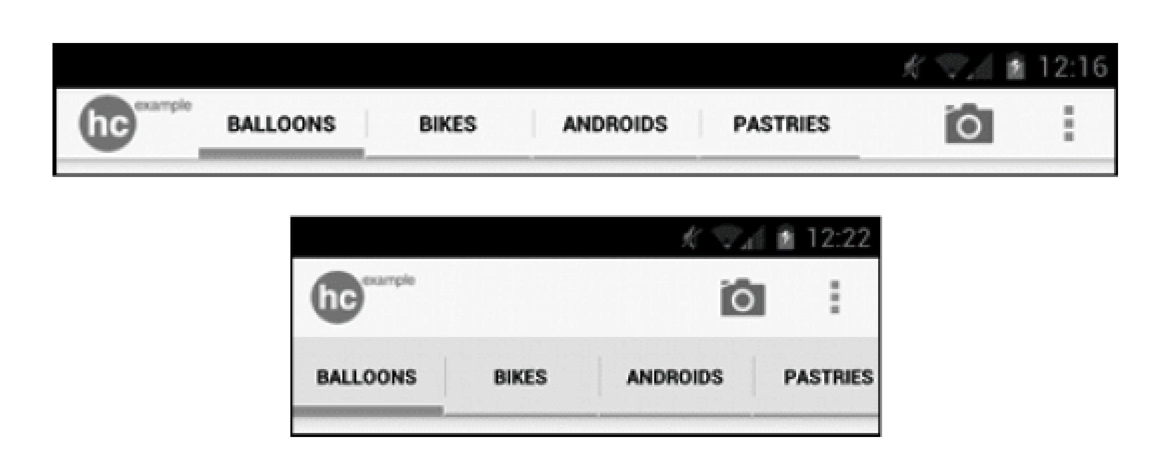


图 4.10 导航标签

要使用选项标签，首先用户界面的布局定义中必须包含一个 ViewGroup 对象，用于放置跟每个 Fragment 对象关联的选项标签。同时这个 ViewGroup 对象需要有一个资源 ID，以便能够在选项标签的切换代码中引用它。如果选项标签的内容填充在 Activity 的布局中，则此 Activity 不需要任何布局，甚至不需要调用 setContentView()方法。如果把每个 Fragment 对象都放到默认的根 ViewGroup 对象中，就用 android.R.id.content ID 来引用这个 ViewGroup 对象。

Fragment 通常作为 Activity 界面的一部分组成出现，并可将它的 Layout 提供给 Activity。Fragment 是应用程序的一部分，可以用 Activity 来替换。一个 Activity 中可以同时出现多个 Fragment，一个 Fragment 亦可在多个 Activity 中使用。在 Activity 运行过程中，可以添加、移除或者替换 Fragment(add()、remove()、replace())。Fragment 可以响应自己的输入事件，并且有自己的生命周期，但直接受其所属的宿主 Activity 的生

命周期影响。

Fragment 在应用中呈现为一个模块化和可重用的组件。将 Fragment 包含到多个 Activity 中，允许应用程序将用户体验适配到不同的屏幕尺寸。例如，当在屏幕尺寸足够大时，在一个 Activity 中可以包含多个 Fragment，相反，应用程序会启动另一个使用不同 Fragment 的 Activity。

决定了 Fragment 对象在布局中的显示位置后，添加选项标签的基本过程如下：

(1) 实现 ActionBar. TabListener 接口。这个接口中回调方法会响应选项标签上的用户事件，例如用户单击，切换 Fragment 对象。

(2) 在初始化 Activity 时，对每个要添加的选项标签，都实例化一个 ActionBar. Tab 对象，并且调用 setTabListener()方法设置 ActionBar. Tab 对象的事件监听器。还可以用 setText()或 setIcon()方法来设置选项标签的标题或图标。

(3) 调用 addTab()方法，将每个选项标签添加到动作条。

ActionBar. TabListener 接口的回调方法并没有说明 Fragment 和 Tab 之间的联系，只提供了被选择的 ActionBar. Tab 对象和执行 Fragment 对象事务的 FragmentTransaction 对象。因此，必须定义每个 ActionBar. Tab 与 Fragment 对象之间的关联。根据设计的不同，有很多种不同的方法来定义这种联系。

下面用一个例子来说明如何实现动作条上的导航标签。

首先，定义实现 ActionBar. TabListener 接口的 Tab 监听器类。在 TabListener 监听器中，定义一个指定 Activity、Fragment 和 Fragment 类型的构造方法，实现 Tab 被选择、被重选和不选几个动作时的事件处理(见代码 4.28)。

代码 4.28 实现 ActionBar. TabListener 接口的 Tab 监听器类

```
public static class TabListener<T extends Fragment> implements ActionBar.
TabListener {
    private Fragment mFragment;
    private final Activity mActivity;
    private final String mTag;
    private final Class<T>mClass;

    /** Constructor used each time a new tab is created.
      * @param activity  The host Activity, used to instantiate the fragment
      * @param tag  The identifier tag for the fragment
      * @param clz  The fragment's Class, used to instantiate the fragment
      */
    public TabListener(Activity activity, String tag, Class<T>clz) {
        mActivity=activity;
        mTag=tag;
        mClass=clz;
    }
    /* The following are each of the ActionBar.TabListener callbacks */
    public void onTabSelected(Tab tab, FragmentTransaction ft) {
```

```
        //Check if the fragment is already initialized
        if(mFragment==null) {
            //If not, instantiate and add it to the activity
            mFragment=Fragment.instantiate(mActivity, mClass.getName());
            ft.add(android.R.id.content, mFragment, mTag);
        } else {
            //If it exists, simply attach it in order to show it
            ft.attach(mFragment);
        }
    }
    public void onTabUnselected(Tab tab, FragmentTransaction ft) {
        if(mFragment !=null) {
            //Detach the fragment, because another one is being attached
            ft.detach(mFragment);
        }
    }
    public void onTabReselected(Tab tab, FragmentTransaction ft) {
        //User selected the already selected tab. Usually do nothing.
    }
}
```

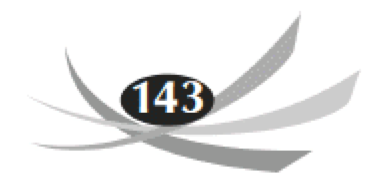
然后,在显示这个动作条的 Activity 的 onCreate()方法中,创建动作条和 ActionBar.Tab 对象,注册 TabListener 监听器到 ActionBar.Tab 对象,并添加到动作条 ActionBar。另外,必须调用 setNavigationMode(ActionBar.NAVIGATION_MODE_TABS)方法来让选项标签可见。如果选项标签的标题实际指示了当前的 View 对象,也可以通过调用 setDisplayShowTitleEnabled(false)方法来禁用 Activity 的标题。代码 4.29 实现了两个导航标签。

代码 4.29　实现导航标签

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //Notice that setContentView() is not used, because we use the root
    //android.R.id.content as the container for each fragment

    //setup action bar for tabs
    ActionBar actionBar=getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
    actionBar.setDisplayShowTitleEnabled(false);

    Tab tab=actionBar.newTab()
            .setText(R.string.artist)
```




```
            .setTabListener(new TabListener<ArtistFragment>(
                    this, "artist", ArtistFragment.class));
    actionBar.addTab(tab);

    tab=actionBar.newTab()
        .setText(R.string.album)
        .setTabListener(new TabListener<AlbumFragment>(
                this, "album", AlbumFragment.class));
    actionBar.addTab(tab);
}
```

在某些情况下，Android系统会把动作条选项标签作为一个下拉列表来显示，以便确保动作条的最优化显示。

上面有关ActionBar. TabListener的实现，只是几种可能的技术之一。在API Demos应用中能够看到更多的这种样式。

4.2.7 应用导航模式

Android系统在3.0之后，提供了更灵活的导航方式。

1. 使用应用图标

应用图标是ActionBar四个区域中的第一个区域，表示应用程序的标识，需要的话可以使用不同的logo或标牌。如果当前不是应用的顶层界面，则在图标左边会有一个向左的箭头，表示“向上”按钮，使用户可以回到上一级界面。默认情况下，应用程序图标显示在动作条的左边，可以作为动作选项来使用。

在这个应用图标上，应用程序可以响应两种操作：

- 返回应用程序的主Activity。
- 向应用程序上级页面导航。

当用户触摸这个图标时，系统会回调Activity的onOptionsItemSelected()方法，响应这个事件，进行运行对应的程序代码。在事件处理代码中，既可以实现主Activity，也可以返回应用程序层次结构中的用户上一步操作界面。

1）返回应用程序的主Activity

如果要通过应用图标的事件响应来返回主Activity，则需在事件处理代码中设置Intent对象中包括FLAG_ACTIVITY_CLEAR_TOP标识。使用这个标识后，如果要启动的Activity在当前任务中已经存在，则堆栈中这个Activity之上的所有Activity都将被销毁，并且把这个Activity显示给用户。

添加这个标识非常重要，否则在响应事件时，系统会再创建一个新的主Activity实例，而不是回退到原有的主Activity，最终可能会造成在当前任务中产生一个很长的拥有多个主Activity的堆栈。添加这个标识，相当于一个堆栈的回退动作，重新调出堆栈中原有的主Activity实例。

代码4.30示例了在onOptionsItemSelected()方法中的事件处理代码，实现了返回应

用程序的主 Activity 的操作。

代码 4.30　返回应用程序的主 Activity

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case android.R.id.home:
            //app icon in action bar clicked; go home
            Intent intent=new Intent(this, HomeActivity.class);
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

为了适应用户从另一个应用程序进入当前 Activity 的情况，还需要添加 Intent 的另一个 FLAG_ACTIVITY_NEW_TASK 标识，使用户在返回主页或上级页面时，系统不会把新的 Activity 添加到当前的任务中，而是在属于自己的应用程序任务中启动。

Android 4.0 之前的版本，默认情况下应用图标就能够作为一个操作项；但从 Android 4.0 版本开始，如果要使用应用图标来返回主页，必须调用 ActionBar 的方法 setHomeButtonEnabled(true)，来设定应用图标能够作为一个操作项。

2）向应用程序上级页面导航

如果要通过应用图标的事件响应来向应用程序上级页面导航，需要通过调用 ActionBar 的 SetDisplayHomeAsUpEnabledtrue（true）方法（见代码 4.31），在 onOptionsItemSelected()实现事件处理。

代码 4.31　向应用程序上级页面导航

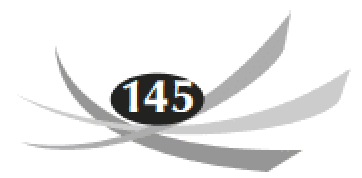
```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
    ActionBar actionBar=getActionBar();
    actionBar.setDisplayHomeAsUpEnabled(true);
    …
}
```

当用户触摸这个图标时，系统会调用带有 android.R.id.home ID 的 onOptionsItemSelected()方法。

2. 添加下拉式导航

作为 Activity 内部的另一种导航模式，动作条提供了内置的下拉列表。例如，下拉列



表能够提供 Activity 中内容的不同排序模式。

启用下拉式导航的基本过程如下：

(1) 创建一个给下拉提供可选项目的列表，以及显示选项时所使用的布局。

(2) 实现 ActionBar. OnNavigationListener 接口，定义用户选项时的事件处理代码。

(3) 在 Activity 的 onCreate()方法中，调用 setNavigationMode()方法，启用动作条的下拉式导航模式。代码如下：

```
ActionBar actionBar=getActionBar();
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
```

(4) 用 ActionBar 的 setListNavigationCallbacks()方法给下拉列表设置回调方法。代码如下：

```
actionBar.setListNavigationCallbacks(mSpinnerAdapter, mNavigationCallback);
```

这个方法需要 SpinnerAdapter 和 ActionBar. OnNavigationListener 对象。

下面使用一个简单的例子来说明如何具体实现下拉式导航(见代码 4.32)。

代码 4.32　实现下拉式导航

```
public class MainActivity extends FragmentActivity implements
    ActionBar.OnNavigationListener {


/**
   * The serialization (saved instance state) Bundle key representing the
   * current dropdown position.
   * /

  private static final String STATE_SELECTED_NAVIGATION_ITEM="selected_
  navigation_item";

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //Set up the action bar to show a dropdown list.
    final ActionBar actionBar=getActionBar();
    actionBar.setDisplayShowTitleEnabled(false);
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);

    final String[] dropdownValues=getResources().getStringArray(R.array.
    dropdown);

    //Specify a SpinnerAdapter to populate the dropdown list.
```

```
    ArrayAdapter<String> adapter=new ArrayAdapter<String>(actionBar.
    getThemedContext(), android.R.layout.simple_spinner_item, android.R.
    id.text1, dropdownValues);

    adapter.setDropDownViewResource(android.R.layout.simple_spinner_
    dropdown_item);

   //Set up the dropdown list navigation in the action bar
   actionBar.setListNavigationCallbacks(adapter, this);

   //Use getActionBar().getThemedContext() to ensure
   //that the text color is always appropriate for the action bar
   //background rather than the activity background
}

@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
   //Restore the previously serialized current dropdown position
   if(savedInstanceState.containsKey(STATE_SELECTED_NAVIGATION_ITEM)) {
     getActionBar().setSelectedNavigationItem(savedInstanceState.getInt
     (STATE_SELECTED_NAVIGATION_ITEM));
   }
}

@Override
public void onSaveInstanceState(Bundle outState) {
   //Serialize the current dropdown position
   outState.putInt(STATE_SELECTED_NAVIGATION_ITEM, getActionBar()
       .getSelectedNavigationIndex());
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
   getMenuInflater().inflate(R.menu.activity_main, menu);
   return true;
}

@Override
public boolean onNavigationItemSelected(int position, long id) {
   //When the given dropdown item is selected, show its contents in the
   //container view
   Fragment fragment=new DummySectionFragment();
   Bundle args=new Bundle();
```

```
    args.putInt(DummySectionFragment.ARG_SECTION_NUMBER, position+1);
    fragment.setArguments(args);
    getFragmentManager().beginTransaction()
        .replace(R.id.container, fragment).commit();
    return true;
  }

/**
   * A dummy fragment
   */

  public static class DummySectionFragment extends Fragment {

    public static final String ARG_SECTION_NUMBER="placeholder_text";

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
      TextView textView=new TextView(getActivity());
      textView.setGravity(Gravity.CENTER);
      textView.setText(Integer.toString(getArguments().getInt(ARG_
      SECTION_NUMBER)));
      return textView;
    }
  }

}
```

4.3 本章小结

本章主要介绍了Android系统的应用浏览模式中菜单和动作条两个最重要的组件。

Android SDK支持丰富的菜单类型，包括常规的菜单、子菜单、上下文菜单、图标菜单、二级菜单和替代菜单。其中选项菜单、弹出菜单和上下文菜单是菜单的三个基本类型。菜单项由android.view.MenuItem类表示，子菜单由android.view.SubMenu类表示。菜单的创建和事件处理与其他视图对象类似，可以通过XML布局文件设计菜单，使用监听器或菜单的回调方法来处理菜单选项事件。

动作条(ActionBar)是位于Activity顶端的一个图形控件，能够显示Activity的标题、图标、可能触发的动作、附加视图和其他交互控件，也可以用于在应用程序中导航。对于大多数应用，动作条可以分割为应用图标、视图控制、动作按钮和溢出动作四个不同的功能区域。

动作条可以与许多控件结合使用，例如菜单、视图、标签和内容提供器等组件，为Android用户界面提供了丰富的显示模式。

第 5 章 发送和接收消息

5.1 理解 Intent

什么是 Intent 呢？Intent 是连接 Android 组件的纽带，专门用于携带需要传递的信息。当某个组件创建一个 Intent 对象并发送后，Android 系统会根据这个 Intent 携带的信息激活对应的其他组件，也就是启动这些组件，执行这些组件的代码。这个 Intent 对象中同时携带触发其他组件执行的条件信息和触发后该组件执行时所需要的信息。

5.1.1 Intent 的概念

由于 Android 是一个基于有限资源的操作系统，Android 的基本设计理念是鼓励减少组件的耦合，因此 Android 提供了 Intent 机制——一种通用的消息系统。Android 的一个应用程序与其他的应用程序之间通过传递 Intent 对象来执行动作和产生事件。通过 Intent 的消息触发、消息传递、消息响应来实现窗口跳转、传递数据或调用外部程序，进行应用程序的激活和调用。

Android 的基础组件 Activity、Service 和 BroadcastReceiver，都可以通过定义 Intent 的消息，实现在各组件之间的程序跳转和数据传递，也就是 Intent 的消息可以激活其他组件。从程序的角度来看，在 Android 中，Intent 相当于各个 Activity 之间或其他类型基础组件的桥梁，可以传递数据，还可以通过 Intent 启动另外一个基础组件。

例如，从一个窗口单击一个链接，用浏览器打开另一个页面时，既要启动浏览器程序，又要把链接传递给浏览器，这时 Android 应用程序就可以在第一个 Activity 中创建一个 Intent 对象，在 Intent 对象把链接的数据封装，然后通过 Android 系统传递给浏览器程序，并启动浏览器。

抽象地说，Intent 消息是同一个应用程序或不同应用程序运行后，组件间进行绑定的一种能力。通过 Intent 消息，把不同的组件与用户的操作联系起来，例如在一个 Activity 上单击一个按钮就打开另一个显示照片的 Activity，而单击链接则打开一个浏览器。

具体来说，Intent 对象包含要执行的操作或需要传递的消息，或者在广播的情况下，包含一些已经发生或正在发生的事情的描述。

举个例子，在一个联系人维护的应用中，当我们在一个联系人列表屏幕（假设对应的 Activity 为 listActivity）上，单击某个联系人后，希望能够跳出此联系人的详细信息屏幕（假设对应的 Activity 为 detailActivity），在两个 Activity 之间，需要传递“联系人”的信息，这个工作由 Intent 完成。

5.1.2　Intent 对象的组成

一个 Intent 对象就是一个信息包。它包含了接收这个 Intent 对象的组件所感兴趣的信息（如要执行的动作和动作相关的数据）和 Android 系统感兴趣的信息（如处理这个 Intent 对象的组件的分类和有关如何启动目标 Activity 的指令）。Intent 为这些不同的信息定义了对应的属性，我们通过设定所需的属性值，就可以把数据从一个 Activity 传递到另一个 Activity。

Intent 对象可绑定的信息如下：

- ComponentName：需要启动的 Activity 的名字。
- Action：指定了要访问的 Activity 需要做什么。
- Data：需要传递的数据。
- Category：给出一些 Action 的额外执行信息。
- Extras：需要传递的额外信息，以键值对形式传递。
- Flags：标记 Activity 启动的方式。

但是 Intent 对象在绑定信息时，并不是所有的信息都必须设置，而只是选定需要携带的信息绑定到 Intent 对象，也就是设定 Intent 对象的对应属性的值。

1. ComponentName

Intent 的组件名称对象由 ComponentName 类封装。也就是说，Intent 定义了一个属性描述 Intent 将要激活或启动的 Android 组件名称，这个组件可以是一个 Activity、Service、BroadcastReceiver 或者 ContentProvider。Intent 的这个属性值是一个 ComponentName 类的对象，我们无法直接访问它，但可以通过 getComponentName() 获取。

ComponentName 类包含两个 String 成员，分别代表 Android 组件的全称类名和包名，包名必须和 AndroidManifest.xml 文件中标记中的对应信息一致。也就是说，这个 Intent 对象所要激活或启动的 Android 组件，已经在 AndroidManifest.xml 中进行了描述。

对于 Intent，组件名并不是必需的。如果一个 Intent 对象添加了组件名，则称该 Intent 为“显式 Intent”，这样的 Intent 在传递时会直接根据组件名去寻找目标组件。如果没有添加组件名，则称为“隐式 Intent”，Android 会根据 Intent 中的其他信息来确定响应该 Intent 的组件。打个比方，老师在让学生回答问题时，老师说：“张三请回答问题”，这就是显式 Intent，直接指出了回答问题的人；老师说：“请第二排第三个同学回答问题”，这就是隐式 Intent，给出了回答问题学生的条件，学生根据自己的座位来确认是谁来回答问题。

2. Action

Action 是描述要求 Android 系统所执行动作的一个属性，值是一个字符串常量，代表 Android 组件所可能执行的一些操作，如启动 Activity、发出警告等。Android 系统中已经预定义了一些 Action 常量，开发者也可以定义自己的 Action 描述。Android 定义了一套标准 Action 的值，其中最重要的和最常用的 Action 操作是 ACTION_MAIN 和

ACTION_EDIT，见表 5.1。

表 5.1 标准的 Action 操作

标准的 Activity Actions	标准的 Broadcast Actions
ACTION_MAIN	ACTION_TIME_TICK
ACTION_VIEW	ACTION_TIME_CHANGED
ACTION_ATTACH_DATA	ACTION_TIMEZONE_CHANGED
ACTION_EDIT	ACTION_BOOT_COMPLETED
ACTION_PICK	ACTION_PACKAGE_ADDED
ACTION_CHOOSER	ACTION_PACKAGE_CHANGED
ACTION_GET_CONTENT	ACTION_PACKAGE_REMOVED
ACTION_DIAL	ACTION_PACKAGE_RESTARTED
ACTION_CALL	ACTION_PACKAGE_DATA_CLEARED
ACTION_SEND	ACTION_UID_REMOVED
ACTION_SENDTO	ACTION_BATTERY_CHANGED
ACTION_ANSWER	ACTION_POWER_CONNECTED
ACTION_INSERT	ACTION_POWER_DISCONNECTED
ACTION_DELETE	ACTION_SHUTDOWN
ACTION_RUN	
ACTION_SYNC	
ACTION_PICK_ACTIVITY	
ACTION_SEARCH	
ACTION_WEB_SEARCH	
ACTION_FACTORY_TEST	

假如这个属性定义了 ACTION_MAIN 动作，则表示接收 Intent 对象传递信息的 Activity 进行初始化和启动操作，并且不需要数据输入也没有返回值输出。

假如这个属性定义了 ACTION_BATTERY_LOW 动作，当电池电量低时，系统会使用 Intent 对象传递这个信息给 BroadcastReceiver 组件。

下面是 Action 和后面携带相关数据的例子：

```
ACTION_VIEW content://contacts/people/1     --显示标识为"1"的联系人信息
ACTION_DIAL content://contacts/people/1     --显示可填写的电话拨号器
ACTION_DIAL tel:123                         --显示带有号码的拨号器
ACTION_EDIT content://contacts/people/1     --编辑标识为"1"的联系人信息
ACTION_VIEW content://contacts/people/      --显示联系人列表
```

在 Java 中使用 setAction()来设置 Intent 的 Action 属性，使用 getAction()来获得 Action 属性。

3. Data

Data 部分描述了 Android 系统执行与 Action 相关的动作时所激活的其他组件执行时需要的数据、数据 MIME 类型和 URI，不同的 Action 对应不同的操作数据。例如，如果动作字段是 ACTION_EDIT，数据字段将包含将显示用于编辑的文档的 URI；如果动

作是 ACTION_CALL,数据字段将是一个 tel:URI 和将拨打的号码;如果动作是 ACTION_VIEW,数据字段是一个 http:URI,接收活动将被调用去下载和显示 URI 指向的数据。

当 Android 系统根据一个 Intent 匹配对应的组件时,通常知道数据的类型(它的 MIME 类型)和它的 URI 很重要。例如,一个能够显示图像数据的组件,就不应该在播放一个音频文件时被激活。

在许多情况下,能够从 URI 中推测数据类型,特别是 content:URIs,它表示位于设备上的数据且被 ContentProvider 控制。但是也能够显示地设置类型,使用 Intent 的 setData()方法指定数据的 URI、setType()指定 MIME 类型,setDataAndType()指定数据的 URI 和 MIME 类型。被激活执行的 Activity 中,获取 Intent 对象后,通过 Intent 的 getData()读取 URI、getType()读取类型。

Data 部分的数据类型是由 Action 的值决定的,下面给出的几个例子可以看出,Action 部分的不同,决定了数据部分的值不同。

```
ACTION_VIEW content://contacts/1    --显示标识符为"1"的联系人的详细信息
ACTION_EDIT content://contacts/1    --编辑标识符为"1"的联系人的详细信息
ACTION_VIEW content://contacts/     --显示所有联系人的列表
ACTION_PICK content://contacts/     --显示所有联系人的列表,并且允许用户在列表
                                      中选择一个联系人,然后把这个联系人返回给
                                      父 Activity
```

4. Category

Category 主要描述被请求组件或执行行为动作的额外信息。Android 系统也为类别定义了一系列的静态常量字符串来表示 Intent 的不同类别,其中标准的类别定义见表 5.2。

表 5.2 标准的 Category 和 Extras

Standard Category	Standard Extra Data
CATEGORY_DEFAULT	EXTRA_ALARM_COUNT
CATEGORY_BROWSABLE	EXTRA_BCC
CATEGORY_TAB	EXTRA_CC
CATEGORY_ALTERNATIVE	EXTRA_CHANGED_COMPONENT_NAME
CATEGORY_SELECTED_ALTERNATIVE	EXTRA_DATA_REMOVED
CATEGORY_LAUNCHER	EXTRA_DOCK_STATE
CATEGORY_INFO	EXTRA_DOCK_STATE_HE_DESK
CATEGORY_HOME	EXTRA_DOCK_STATE_LE_DESK
CATEGORY_PREFERENCE	EXTRA_DOCK_STATE_CAR
CATEGORY_TEST	EXTRA_DOCK_STATE_DESK
CATEGORY_CAR_DOCK	EXTRA_DOCK_STATE_UNDOCKED

续表

Standard Category	Standard Extra Data
CATEGORY_DESK_DOCK CATEGORY_LE_DESK_DOCK CATEGORY_HE_DESK_DOCK CATEGORY_CAR_MODE CATEGORY_APP_MARKET	EXTRA_DONT_KILL_APP EXTRA_EMAIL EXTRA_INITIAL_INTENTS EXTRA_INTENT EXTRA_KEY_EVENT EXTRA_ORIGINATING_URI EXTRA_PHONE_NUMBER EXTRA_REFERRER EXTRA_REMOTE_INTENT_TOKEN EXTRA_REPLACING EXTRA_SHORTCUT_ICON EXTRA_SHORTCUT_ICON_RESOURCE EXTRA_SHORTCUT_INTENT EXTRA_STREAM EXTRA_SHORTCUT_NAME EXTRA_SUBJECT EXTRA_TEMPLATE EXTRA_TEXT EXTRA_TITLE EXTRA_UID

5. Extras

主要描述组件的扩展信息或额外的数据。Android 也定义了标准的 Extra Data 常量,见表 5.2。

Extras 采用键值对的结构,以 Bundle 对象的形式保存在 Intent 当中。附加信息其实是一个类型安全的容器,其实现就是将 HashMap 做了一层封装。

Intent 对象有一系列的 put…()方法用于插入各种附加数据和一系列的 get…()方法用于读取数据。这些方法与 Bundle 对象的方法类似。实际上,Extras 可以作为一个 Bundle,使用 putExtras()和 getExtras()方法安装和读取。

例如,如果要执行“发送电子邮件”这个动作,可以将电子邮件的标题、正文等保存在 Extras 里,传给电子邮件发送组件。

6. Flags

Flags 主要标示如何触发目标组件以及如何看待被触发的目标组件。例如标示被触发的组件应该属于哪一个任务或者触发的组件是否是最近的 Activity 等。Flags 可以是多个标示符的组合。

Android 有各种各样的标志,指示 Android 系统如何去启动一个 Activity(例如,Activity 应该属于那个任务)和启动之后如何对待它(例如,它是否属于最近的活动列表)。所有这些标志都定义在 Intent 类中。其可用的常量包括 FLAG_ACTIVITY_CLEAR_TOP、FLAG_ACTIVITY_NEW_TASK、FLAG_ACTIVITY_NO_HISTORY、FLAG_ACTIVITY_SINGLE_TOP。

5.1.3 Intent解析

当一个Activity创建并发出一个Intent对象后，其他的Activity或基础组件可能因为与这个Intent对象携带的信息相关而被启动。一个Activity或其他基础组件实现在各组件之间的程序跳转和数据传递，可以通过Intent的两种不同方式来实现：显式和隐式。

(1) 显式Intent：指在Activity或其他组件创建Intent对象时，通过Component Name显式地设定所希望启动、激活或接收这个Intent对象的目标组件(如Service)名称，当这个Activity发送这个Intent对象后，由Android系统自动启动或激活Component Name指定的Android组件，接收这个Intent对象携带的其他信息。因为有时开发者不知道其他应用的Component名称，显式方式常用于自己应用内部的消息传递，比如应用中一个Activity启动一个相关的Service或者启动一个相关的Activity。

(2) 隐式Intent：创建Intent对象的组件，并不指定目标组件的名字，即Component Name字段为空，当这个Activity发送这个Intent对象后，Android系统通过Intent对象中的其他信息与目标组件的Intent过滤器中的设置相匹配，来启动或激活相关的Activity、Service或BroadcastReceiver等目标组件。隐式Intent经常用于激活其他应用程序中的组件。

对于接收隐式的Intent的Android组件来说，需要在AndroidManifest.xml中设定接收Intent对象的策略。当Android系统在处理Intent对象时，把Intent对象中携带的信息与应用程序中组件设定的Intent过滤策略逐个相比较，判断该组件是否符合启动或接收的条件。也就是说，通过设定Intent过滤策略条件，可以指示Android系统在什么时候启动并把Intent对象传递给自己。

下面是一个使用Intent Filter的例子。

```
<activity android:name=".IntentExampleActivity">
    <intent-filter>
        <action android:name="com.android.activity.MY_ACTION"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

这是在AndroidManifest.xml文件中IntentExampleActivity的声明代码。当系统中其他的Activity发送出一个Intent对象，并且这个Intent对象的Action属性的值为"MY_ACTION"时，Android系统会启动IntentExampleActivity应用程序。

如果一个组件没有声明任何Intent过滤器，它仅能接收显式的Intent，也就是被显式Intent对象启动或激活；而声明了Intent过滤器的组件可以接收显式和隐式的Intent。

并非Intent对象中所有的信息都会用于过滤器的匹配，只有Action、Data(包括URI和数据类型)、Category三个字段才被考虑。

下面具体讨论一下Intent过滤器及其检测方法。

1. Intent过滤器

Intent过滤器是由intent-filter标记来设置的。Activity、Service、BroadcastReceiver

为了告知 Android 系统能够处理哪些隐式 Intent，可以设置一个或多个条件，说明该组件可接收的 Intent 对象，过滤掉不想接收的隐式 Intent 对象。

除了 BroadcastReceiver 通过调用 Context. registerReceiver()动态地注册，直接创建一个 Intent Filter 对象外，其他的 Intent 过滤器必须在 AndroidManifest. xml 文件中进行声明。可用于 Intent 过滤器的 Intent 字段有三个：Action、Data 和 Category。

在 Android 系统通过组件的 Intent 过滤器检测隐式 Intent 时，要检测所有这三个字段，其中任何一个字段匹配失败，系统都不会把这个隐式 Intent 给该组件。但每个字段可以用 intent-filter 设置多个条件，每个设定的条件之间相互独立，只要其他组件发送出的隐式 Intent 对象符合其中的一个条件，就能够被 Android 系统启动或接收。

2. Action 检测

在 AndroidManifest. xml 文件中，对 Action 字段设置过滤条件，在<intent-filter>元素下使用<action>子元素及其属性 android:name 来设置可接收的 Action 字段的值。例如：

```
<intent-filter>
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
</intent-filter>
```

根据例子设置的过滤策略，只要 Action 字段的值符合上面列出之一的 Intent，都可以被这个组件接收。虽然一个 Intent 对象的 Action 只有一个值，但是一个过滤器可以列出不止一个，接收多种类型 Action 的 Intent 对象。

值得注意的是，<intent-filter>元素下必须至少包含一个<action>子元素，否则它将阻塞所有的 intents。要通过检测，Intent 对象中指定的动作必须匹配 Intent 过滤器的 Action 列表中的一个。如果过滤器没有<action>子元素，将没有一个 Intent 匹配，所有的 Intent 都会检测失败，没有 Intent 能够通过过滤器。

3. Category 检测

类似 Action 检测，在<intent-filter>元素下使用<category>子元素及其属性 android:name 列出可接收的 Category 字段的值。例如：

```
<intent-filter …>
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    …
</intent-filter>
```

在 Intent 对象中可以含有多个 Category，<intent-filter>中可以设置多个<category>，只有 Intent 中的所有 Category 都能匹配到<intent-filter>中的<category>，Intent 才能通过检测。也就是说，如果 Intent 对象中的 Category 集合是<intent-filter>中<category>的集合的子集时，Intent 对象才能通过检查。如果 Intent 对象中没有设

置 Category 的值，则它能通过所有<intent-filter>的<category>检查。

如果一个 Intent 能够通过不止一个组件的<intent-filter>，系统可能会询问哪个组件被激活。如果找不到目标组件，会产生一个异常。

4. Data 检测

类似的，在<intent-filter>元素下使用<data>元素及其属性可接收的 Data 字段的值。例如：

```
<intent-filter …>
    <data android:mimeType="video/mpeg" android:scheme="http" … />
    <data android:mimeType="audio/mpeg" android:scheme="http" … />
    …
</intent-filter>
```

每个<data>元素可以指定一个 URI 和数据类型(MIME 类型)。对应于 URI 的 scheme、host、port、path 四个部分，<data>元素分别使用属性 android:scheme、android:host、android:port、android:path 来设置。下面是 URI 的格式和一个例子。

```
scheme://host:port/path
content://com.example.project:200/folder/subfolder/etc
```

scheme 是 content，host 是"com. example. project"，port 是 200，path 是"folder/subfolder/etc"。host 和 port 一起构成 URI 的凭据(authority)，如果 host 没有指定，port 也被忽略。这四个属性都是可选的，但它们之间并不都是完全独立的。要让 authority 有意义，scheme 必须也要指定。要让 path 有意义，scheme 和 authority 也都必须要指定。

当比较 Intent 对象和过滤器的 URI 时，仅仅比较过滤器中出现的 URI 属性。例如，如果一个过滤器仅指定了 scheme，所有有此 scheme 的 URI 都匹配过滤器；如果一个过滤器指定了 scheme 和 authority，但没有指定 path，所有匹配 scheme 和 authority 的 URI 都通过检测，而不管它们的 path；如果四个属性都指定了，要都匹配才能算是匹配。然而，过滤器中的 path 可以包含通配符来要求匹配 path 中的一部分。

<data>元素的 mimeType 属性指定数据的 MIME 类型。Intent 对象和过滤器都可以用"*"通配符匹配子类型字段，例如"text/*"，"audio/*"表示任何子类型。

在 Data 检测时，系统既要检测 URI，也要检测 MIME 类型。检测的规则如下：

(1) 一个 Intent 对象既不包含 URI，也不包含 MIME 类型：仅当过滤器也不指定任何 URI 和 MIME 类型时，才不能通过检测；否则都能通过。

(2) 一个 Intent 对象包含 URI，但不包含 MIME 类型：仅当过滤器也不指定 MIME 类型，同时它们的 URI 匹配，才能通过检测。例如，mailto:和 tel:都不指定实际数据。

(3) 一个 Intent 对象包含 MIME 类型，但不包含 URI：仅当过滤器也只包含 MIME 类型且与 Intent 相同，才通过检测。

(4) 一个 Intent 对象既包含 URI，也包含 MIME 类型(或 MIME 类型能够从 URI 推断出)：MIME 类型部分，只有与过滤器中之一匹配才算通过；URI 部分，它的 URI 要出

现在过滤器中，或者它有 content：或 file：URI，又或者过滤器没有指定 URI。换句话说，如果它的过滤器仅列出了数据类型，组件假定支持 content：和 file：。

如果一个 Intent 能够通过不止一个组件的<intent-filter>，系统可能会询问哪个组件被激活。如果找不到目标组件，会产生一个异常。

例如，Android 一个名为 Note Pad 的应用程序，允许用户浏览便签，查看每条便签的内容。这个程序在 Manifest 文件中可以按代码 5.1 来设置 Intent 过滤器，使其可以根据需要被系统激活。

代码 5.1　应用程序的 Intent 过滤器设置

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.android.notepad">
    <application android:icon="@drawable/app_notes"
          android:label="@string/app_name">

        <provider class=".NotePadProvider"
              android:authorities="com.google.provider.NotePad" />

        <activity class=".NotesList" android:label="@string/title_notes_
        list">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"
                </category>
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT"
                />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.
                note" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.GET_CONTENT"
                </action>
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.item/vnd.
                google.note" />
            </intent-filter>
        </activity>

        <activity class=".NoteEditor" android:label="@string/title_
        note">
```

```
            <intent-filter android:label="@string/resolve_edit">
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.item/vnd.
                google.note" />
            </intent-filter>

            <intent-filter>
                <action android:name="android.intent.action.INSERT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.
                note" />
            </intent-filter>

        </activity>

        <activity class=".TitleEditor" android:label="@string/title_edit_
        title" android:theme="@android:style/Theme.Dialog">
            <intent-filter android:label="@string/resolve_title">
                <action android:name="com.android.notepad.action.EDIT_
                TITLE" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.
                ALTERNATIVE" />
                <category android:name="android.intent.category.SELECTED_
                ALTERNATIVE" />
                <data android:mimeType="vnd.android.cursor.item/vnd.
                google.note" />
            </intent-filter>
        </activity>

    </application>
</manifest>
```

在代码5.1中，可知此应用程序定义了三个Activity，每一个Activity都定义了多个Intent模板。其中命名为com.android.notepad.NotesList的第一个Activity作为进入应用程序的主入口，通过定义三个Intent过滤器，可以做三件事情。

第一个Intent过滤模板代码如下：

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

Action 设置为标准 MAIN，表示这个 Activity 提供了进入 NotePad 应用程序的顶级入口，Category 设置为 LAUNCHER 表示这个入口应该列入应用程序启动列表。

第二个 Intent 过滤模板代码如下：

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data mimeType:name="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

Action 设置为 VIEW、EDIT 和 PICK 表示这个 Activity 可以对便签目录所作的操作，允许用户浏览、编辑和挑选便签。Category 设置 DEFAULT 表示如果这个 Activity 的组件名没有显示说明，还需要通过 Context. startActivity()方法来启动这个 Activity。

第三个 Intent 过滤模板代码如下：

```
<intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

vnd. android. cursor. item/vnd. google. note 是指示 vnd. android. cursor. item 资源中确切指定的一个 URI，也就是 vnd. google. note。Data 的 type 的设置表示指定类型的数据可以被这个 Activity 检索。Action 设置为 GET_CONTENT 与 PICK 类似。这个设置表示当 type 为 vnd. android. cursor. item/vnd. google. note 时，返回给调用者一个用户选择的便笺，而用户不需要知道便笺是从哪里读取的。

通过这三个过滤模板的设置，如果系统中出现携带下面信息的 Intent，NotesList 这个 Activity 就会被激活执行。

（1）{action＝android. app. action. MAIN}：与此 Intent 匹配的 Activity，将会被当作进入应用的顶级入口。

（2）{action ＝ android. app. action. MAIN，category ＝ android. app. category. LAUNCHER}：这是目前 Launcher 实际使用的 Intent，用于生成 Launcher 的顶级列表。

（3）{action＝android. app. action. VIEW，data＝content://com. google. provider. NotePad/notes}：显示"content://com. google. provider. NotePad/notes"下的所有便笺的列表，使用者可以遍历列表，并且查看某便笺的详细信息。

（4）{action＝android. app. action. PICK，data＝content://com. google. provider. NotePad/notes}：显示"content://com. google. provider. NotePad/notes"下的便笺列表，让用户可以在列表中选择一个，然后将选择的便笺的 URL 返回给调用者。

```
{action=android.app.action.GET_CONTENT, type=vnd.android.cursor.item/
vnd.google
```

下面我们使用实际的例子来创建Intent，使用显式和隐式Intent激活其他组件，加深对Intent组件的理解，了解如何利用Intent在Activity之间传递数据。

5.1.4 使用Intent实现数据传递

使用Intent实现数据传递无论是显式还是隐式，都需要有以下几个步骤：

(1) 定义传递数据的Activity，也就是通过布局文件设计Activity的界面，并创建相互转换的几个Activity，它们之间需要数据转换或需要在某种情况下进行切换。

(2) 在Activity中创建Intent，设定所传递元素的值，即需传递的数据；或者设定所要切换的Activity。

(3) 声明Activity以及Intent Filter，这一步在manifest.xml中声明所创建的Activity，并且根据显式还是隐式的设置设定相应的Intent Filter。

下面我们使用例子说明Intent如何在Activity之间起作用。

1. 使用显式定义Intent

使用显式定义Intent，实现About对话框功能：

- 在主页面，用户单击About按钮时，弹出一个对话框，显示有关移动电子商务平台的信息。
- 在About对话框中，单击OK按钮，返回主页面。

下面先给出关键的Intent创建代码，方便在完整的程序代码中查看。这里直接给出了Intent的组件名称。

```
//显式方式声明Intent,直接启动SecondActivity
Intent it=new Intent(MainActivity.this,SecondActivity.class);
//启动Activity
startActivity(it);
```

完成这个功能，需要几个步骤：

(1) 定义主Activity的布局，设置一个显示信息的文本框TextView和About按钮，见代码5.2。

代码5.2 explicit_intent_main_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"    >
    <TextView
     android:layout_width="fill_parent"
```

```
     android:layout_height="wrap_content"
     android:text="@string/hello"    />
    <Button
     android:id="@+id/btn"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/about_button"    />
</LinearLayout>
```

(2) 定义单击按钮后出现的第二个 Activity 的布局，设置一个显示信息的文本框 TextView 和 OK 按钮，见代码 5.3。

代码 5.3　explicit_intent_second_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"    >
    <TextView
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="@string/about_button_show"    />
    <Button
     android:id="@+id/secondBtn"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="OK"    />
</LinearLayout>
```

(3) 定义主 Activity，导入布局资源定义的界面，并在 OnClickListener() 方法中编写单击按钮后的事件处理代码，见代码 5.4。

代码 5.4　ExplicitMainActivity.java

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class ExplicitMainActivity extends Activity {
    private Button btn;
    @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. explicit_intent_main);

        btn=(Button)findViewById(R.id.btn);
        //响应按钮 btn 事件
        btn.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                //显式方式声明 Intent,直接启动 SecondActivity
                Intent it=new Intent(MainActivity.this,SecondActivity.
                class);
                //启动 Activity
                startActivity(it);
            }
        });
    }
}
```

(4) 定义第二个用户界面 Activity，导入布局资源定义的界面，并在 OnClickListener() 方法中编写单击 OK 按钮后的返回主界面的事件处理代码，见代码 5.5。

代码 5.5　ExplicitSecondActivity.java

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class ExplicitSecondActivity extends Activity {
    private Button secondBtn;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. explicit_intent_second);

        secondBtn=(Button)findViewById(R.id.secondBtn);
        //响应按钮 secondBtn 事件
        secondBtn.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                //显式方式声明 Intent,直接启动 MainActivity
```

```
                Intent intent=new Intent(SecondActivity.this,MainActivity.
                class);
                //启动 Activity
                startActivity(intent);
            }
        });
    }
}
```

完成上述四步后，在 AndroidManifest.xml 中注册，运行此程序，可以看到通过显式 Intent 的作用，可以在两个界面之间跳转。通过这个例子可以看出，所谓显式 Intent，就是通过创建 Intent 对象，直接告诉系统要启动哪一个 Activity 或其他组件。

2. 使用隐式定义 Intent

在上面的例子中，使用显式 Intent 实现了界面的跳转。下面尝试使用隐式 Intent 来实现与上述例子同样的功能，学习 Action 检测的使用

下面先给出关键的 Intent 创建代码，方便在完整的程序代码中查看。

```
//实例化 Intent
Intent it=new Intent();
//设置 Intent 的 Action 属性——自定义的 Action
it.setAction("com.android.activity.MY_ACTION");
//启动 Activity
startActivity(it);
```

在 Androidmanifest.xml 中的主界面声明中的代码如下：

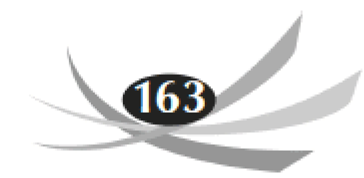
```
<activity android:name=".ImplicitSecondActivity">
    <intent-filter>
        <action android:name="com.android.activity.MY_ACTION"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

完成这个功能，需要几个步骤：

(1) 定义主 Activity 的布局，设置一个显示信息的文本框 TextView 和 About 按钮，见代码 5.2。

(2) 定义单击按钮后出现的第二个 Activity 的布局，设置一个显示信息的文本框 TextView 和 OK 钮，见代码 5.3。

(3) 定义主 Activity，导入布局资源定义的界面，并在 OnClickListener() 方法中编写单击按钮后的事件处理代码，创建 Intent 对象，设置 Intent 对象的 Action 字段的值为 com.android.activity.MY_ACTION，见代码 5.6。



代码 5.6　ImplicitMainActivity.java

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class ImplicitMainActivity extends Activity {
    private Button btn;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. implicit_intent_second_layout);

        btn=(Button) findViewById(R.id.btn);
        //响应按钮 btn 事件
        btn.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                //实例化 Intent
                Intent it=new Intent();
                //设置 Intent 的 Action 属性
                it.setAction("com.android.activity.MY_ACTION");
                //启动 Activity
                startActivity(it);
            }
      });
    }
}
```

(4) 定义第二个用户界面 Activity,导入布局资源定义的界面,见代码 5.7。

代码 5.7　ImplicitSecondActivity.java

```
import android.app.Activity;
import android.os.Bundle;

public class ImplicitSecondActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout. implicit_intent_second_layout);
    }
}
```

3. 使用 Intent 传递和获取数据

无论是显式 Intent 还是隐式 Intent，除了可以启动和激活其他的组件之外，还可以同时携带需要传递到另一个组件的信息，如在一个 Activity 中提供给用户输入界面，完成的多个信息输入，然后启动另一个 Activity，在第二个 Activity 上进行信息的处理和输出。要实现这种功能，可以在第一个 Activity 创建 Intent 对象，设置 Intent 的启动条件，也就是设置显式 Intent 或隐式 Intent 的过滤条件，同时把用户输入的信息也放入这个 Intent。这样在第二个 Activity 接收这个 Intent 时，就可以从 Intent 对象中读出用户输入的信息了。类似的信息可以在 Intent 对象的 Extra 字段中存储。

下面这个例子实现在 FirstActivity 和 SecondActivity 之间传递用户输入信息的功能：

- 在主页面，用户单击 About 按钮时，弹出一个对话框，显示有关移动电子商务平台的信息。
- 在 About 对话框中，单击 OK 按钮，返回主页面。

代码 5.8 给出 Intent 对象设置的主要代码，可以按照前面隐式的 Intent 使用方式来编写布局文件和完整的应用程序。

代码 5.8　Intent 传递和获取数据

```
FirstActivity:
        Intenti=newIntent(this,ActivitySecond.class);
        i.putExtra("Value1","ThisvalueoneforActivityTwo");
        i.putExtra("Value2","ThisvaluetwoActivityTwo");
        startActivityForResult(i,REQUEST_CODE);

SecondActivity:
        Bundleextras=getIntent().getExtras();
        Stringvalue1=extras.getString("Value1");
        Stringvalue2=extras.getString("Value2");
```

4. 通过 Intent 启动系统应用 Activity

通过 Intent 不仅可以启动本项目中的应用程序，还可以通过不同的设定，启动系统提供的应用程序，利用系统定义的功能。具体的启动和激活方式，可以使用显式 Intent，也可以使用隐式 Intent 的 Action、Category 和 Data 的任意一种过滤条件设置。

例如，在下面的代码中定义了一个 URI，并把这个 URI 作为新创建的 Intent 对象的 Data，并将 Intent 的 Action 设置为 ACTION_VIEW。通过这样的设置，当前的 Activity 发送出这个 Intent 之后，就可以启动系统的浏览器，并通过浏览器打开 http://open.taobao.com 这个链接的网页。

```
Uri uri=Uri.parse("http://open.taobao.com");
Intent it=new Intent(Intent.ACTION_VIEW,uri);
startActivity(it);
```

下面的例子中定义了一个单选列表界面，简单调用浏览器、电话拨号、日历等系统应用程序。通过这个例子，可以了解如何利用 Intent 启动和激活常用的系统应用程序。

首先定义应用程序的界面布局（见代码 5.9），然后创建用户界面 Activity（见代码 5.10）。

代码 5.9　to_system_intent _layout. xml

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:alignmentMode="alignBounds"
android:columnCount="1">

    <RadioGroup
     android:id="@+id/action"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:checkedButton="@+id/RadioButton0"
     android:orientation="vertical">

        <RadioButton
         android:id="@+id/RadioButton0"
         android:text="@string/radio_button_0" />

        <RadioButton
         android:id="@+id/RadioButton1"
         android:text="@string/radio_button_1" />

        <RadioButton
         android:id="@+id/RadioButton2"
         android:text="@string/radio_button_2" />

        <RadioButton
         android:id="@+id/RadioButton3"
         android:text="@string/radio_button_3" />

        <RadioButton
         android:id="@+id/RadioButton4"
```

```
        android:text="@string/radio_button_4" />

       <RadioButton
        android:id="@+id/RadioButton5"
        android:text="@string/radio_button_5" />

       <RadioButton
        android:id="@+id/RadioButton6"
        android:text="@string/radio_button_6" />

       <RadioButton
        android:id="@+id/RadioButton7"
        android:text="@string/radio_button_7" />
    </RadioGroup>

    <Button
     android:id="@+id/trigger"
     android:onClick="onClick"
     android:text="Trigger Intent">
    </Button>

</GridLayout>
```

代码 5.10　ToSystemActivity.java

```
import com.taobao.mcommerce.sample.R;

import android.app.Activity;
import android.content.Intent;
import android.content.res.Resources;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.*;

public class CallIntentsActivity extends Activity {
    private Spinner spinner;
    private RadioGroup mRadioGroup;

    /** Called when the activity is first created. */

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.implicit_intent_layout);
        mRadioGroup=(RadioGroup) findViewById(R.id.action);
    }

    public void onClick(View view) {
        int position=mRadioGroup.getCheckedRadioButtonId();
        Intent intent=null;
        switch(position) {
        case R.id.RadioButton0:
            intent=new Intent(Intent.ACTION_VIEW,
                    Uri.parse("http://open.taobo.com"));
            break;
        case R.id.RadioButton1:
            intent=new Intent(Intent.ACTION_CALL,
                    Uri.parse("tel:(+010)1234578"));
            break;
        case R.id.RadioButton2:
            intent=new Intent(Intent.ACTION_DIAL,
                    Uri.parse("tel:(+010)1234578"));
            startActivity(intent);
            break;
        case R.id.RadioButton3:
            intent=new Intent(Intent.ACTION_VIEW,
                    Uri.parse("geo:50.123,7.1434"));
            break;
        case R.id.RadioButton4:
            intent=new Intent(Intent.ACTION_VIEW,
                    Uri.parse("geo:0,0?q=query"));
            break;
        case R.id.RadioButton5:
            intent=new Intent("android.media.action.IMAGE_CAPTURE");
            break;
        case R.id.RadioButton6:
            intent=new Intent(Intent.ACTION_VIEW,
                    Uri.parse("content://contacts/people/"));
            break;
        case R.id.RadioButton7:
            intent=new Intent(Intent.ACTION_EDIT,
                    Uri.parse("content://contacts/people/1"));
            break;

        }
```

```
        if(intent !=null) {
            startActivity(intent);
        }
    }

    @Override
    public void onActivityResult (int requestCode, int resultCode, Intent
    data) {
        if(resultCode==Activity.RESULT_OK && requestCode==0) {
            String result=data.toURI();
            Toast.makeText(this, result, Toast.LENGTH_LONG);
        }
    }
}
```

前面只列出了主要的布局文件代码和 Activity 定义的代码，如果要使程序顺利、正常执行，还需要定义字符串等资源，Activity 也需要在 AndroidMenifest.xml 文件中注册。

通过前面的例子，我们讨论了如何运用 Intent 对象启动和激活其他的组件，如何在组件之间发送和接收消息。在应用程序设计中，可以根据需要灵活运用 Intent 的各种功能。

5.2 BroadcastReceiver 组件

5.2.1 BroadcastReceiver 的概念

Broadcast 是一种广泛运用的在应用程序之间传输信息的机制。BroadcastReceiver 是 Android 系统中负责接收广播消息并对消息做出反应的组件。可以将 BroadcastReceiver 理解为广播接收者，用于接收程序所发出的承载各种各样广播消息的 Intent。它在本质上相当于一个监听器，监听接收广播消息，然后再做出处理。广播消息既可以是系统发送，也可以由用户应用程序产生。

多数的广播是系统发送的，如地域变换、电量不足、来电来信等。程序也可以播放一个广播。BroadcastReceiver 没有用户界面，可以接收到信息后启动 Activity 或者通过 notificationManger 通知用户，也可以通过其他多种方式通知用户，例如开启背景灯、振动设备、播放声音等，最典型的是在状态栏显示一个图标，这样用户就可以单击它打开看通知内容。

如果是用户应用程序发送广播消息，在 Intent 对象创建后，启动 BroadcastRecevicer 的方式有两种：通过 sendBroadcast()方法启动和通过 sendOrderedBroadcast()方法启动。这两者的区别就是前者是发送一个普通的广播，后者是发送一个有序的广播。

BroadcastReceiver 在 Android 应用程序中，与其他三大组件 Activity、Service 和 ContentProvider 一样，是以一段独立的 Java 程序代码存在于应用程序项目中，如果要在

程序中能够启动和运行，必须要在 Android 项目中注册。

注册 BroadcastReceiver 有两种方式：静态注册和动态注册。

静态注册：在 AndroidManifest.xml 中用标签声明注册，并在标签内用标签设置过滤器。例如，静态注册命名为 myRecevicer 的 BroadcastReceiver 的代码如下：

```
<receiver android:name="myRecevicer">
    <intent-filter>
        <action android:name="com.dragon.net"></action>
    </intent-filter>
</receiver>
```

除上述静态的注册方式之外，BroadcastReceiver 也可以通过动态的方式注册，通过 Context.registerReceiver() 方法来实现。动态实现注册的类必须是 BroadcastReceiver 的子类。对应于静态注册例子的动态代码如下：

```
IntentFilter intentFilter=new IntentFilter();
intentFilter.addAction(String);
registerReceiver(BroadcastReceiver,intentFilter);
```

如果要使用 BroadcastReceiver 的功能，首先在需要发送信息的地方创建 Intent 对象，把要携带的信息和用于过滤的信息载入 Intent 对象中，然后通过调用 sendOrderBroadcast() 或 sendStickyBroadcast()方法，把 Intent 对象以广播方式发送出去。

当 Intent 发送以后，所有已经注册的 BroadcastReceiver 会检查注册时的 Intent Filter 是否与发送的 Intent 相匹配，若匹配则就会调用 BroadcastReceiver 的 onReceive() 方法。所以当我们定义一个 BroadcastReceiver 的时候，都需要实现 onReceive()方法。

Android 系统中定义了很多标准的 Broadcast Action 来响应系统的广播事件，见表 5.1。需要的话，这些 Action 可以在应用程序中直接赋给 Intent 的 Action 字段。

下面我们分别通过简单的例子来学习在应用程序中其他组件如何创建和使用 Broadcast 消息，如何使用不同的注册方式来设置 BroadcastReceiver 的过滤条件、处理广播信息，如何使用 BroadcastReceiver 来处理系统广播信息。

5.2.2　静态注册方式

BroadcastReceiver 在 AndroidManifest.xml 文件中进行静态注册，使用<application>元素的子元素<receiver>注册要使用的 BroadcastReceiver，并在<receiver>的子元素<intent-filter>中定义过滤条件，确定接收处理哪一类的 Intent。

如果 BroadcastReceiver 采用静态方式注册，无论该项目的应用程序是否处于活动状态，都会进行监听。如某个程序设定监听电池使用情况的 BroadcastReceiver，当程序在手机上安装好后，不管这个应用程序处于什么状态，当收到系统广播的或其他应用程序广播的有关电池使用状况的的内容，都会执行其 onReceive()中的内容。

下面的例子定义了一个 BroadcastReceiver，当其接收到一个广播消息，即应用程序发

送出的一个广播 Intent，它的 Action 字段是 RECEIVER_ACTION 时，对这个广播消息事件进行处理，在其 onRecieve()方法中把这个 Intent 的有关信息写入日志。

要实现这个例子的功能，需要做一些工作：

(1) 定义发送广播 Intent 的界面，包括布局文件 main. xml 和 Activity。

(2) 定义 BroadcastReceiver，处理广播消息。

(3) 在 AndroidManifest. xml 中注册所定义的 Broadcast Receiver，定义其过滤器。

首先定义用户发送广播界面的布局文件 main. xml（见代码 5. 11），在界面上定义一个按钮，使其显示出如图 5. 1 所示的界面。



图 5. 1　发送 Broadcast 界面

代码 5. 11　main. xml

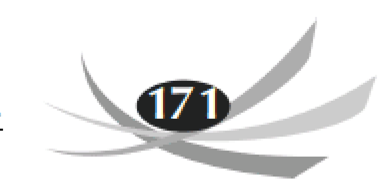
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button
        android:id="@+id/btnBroadcast"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="发送 Broadcast"
        />
</LinearLayout>
```

图 5. 1 所示的界面是这个例子的用户主界面，其中的按钮用于控制广播消息 Intent 的创建和发送。在定义界面的 Activity 中导入布局文件后，在按钮事件处理代码中创建一个 Intent，进行 Action 设置，并使用 sendBroadcast()发送这个 Intent，产生一个广播消息，见代码 5. 12。

代码 5. 12　TestStaticReceiverActivity. java

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class TestStaticReceiverActivity extends Activity{
    //定义 Action 常量
    protected static final String ACTION="com.android.broadcast.RECEIVER_
    ACTION";
```




```
    //定义 Button 对象
    private Button btnBroadcast;
    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        btnBroadcast= (Button)findViewById(R.id.btnBroadcast);
        //为按钮设置单击监听器
        btnBroadcast.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View v){
                //实例化 Intent
                Intent intent=new Intent();
                //设置 Intent 的 Action 属性
                intent.setAction(ACTION);
                //发出广播
                sendBroadcast(intent);
            }
        });
    }
}
```

完成创建和发送广播消息的功能之后，需要定义一个 BroadcastReceiver 来处理所发送的广播消息，实现广播日志的填写。

定义一个 MyReceiver 类，继承于 BroadcastReceiver，覆盖 onReceive()方法，在其中实现写日志操作，见代码 5.13。

代码 5.13　MyReceiver.java

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class MyReceiver extends BroadcastReceiver{
    //定义日志标签
    private static final String TAG="Test";
    @Override
    public void onReceive(Context context, Intent intent){
        //输出日志信息
        Log.i(TAG, "MyReceiver onReceive--->");
    }
}
```

定义创建广播消息的 Activity 和处理广播消息的 BroadcastReceiver 之后，就可以完

成在 AndroidManifest.xml 配置文件中的静态注册了。

在 AndroidManifest.xml 文件的<application>元素下，声明 TestBroadcastActivity 和 MyReceiver。在 MyReceiver 的<intent-filter>中设定"com.android.broadcast.RECEIVER_ACTION"为符合接收条件 action 字段的值，见代码 5.14。

代码 5.14　AndroidManifest.xml 配置文件

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.android.broadcast"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="10" />

<application android:icon="@drawable/icon" android:label="@string/app_
name">
        …
        <activity android:name=".TestStaticReceiverActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="MyReceiver">
            <intent-filter>
                <action android:name="com.android.broadcast.RECEIVER_
                ACTION"/>
            </intent-filter>
        </receiver>
        …
    </application>
</manifest>
```

执行应用程序，单击用户主界面图 5.1 中的按钮，程序会调用 MyReceiver 中的 onReceive()方法，LogCat 输出信息见图 5.2。

Time		pid	tag	Message
09-01 13:17...	I	863	Test	MyReceiver onReceive-->

图 5.2　logCat 输出信息

5.2.3　动态注册方式

BroadcastReceiver 的动态注册方式采用在 Java 程序代码中调用 Activity 的方法来

注册。因为是在程序运行过程中才注册，所以称为动态注册。注册时，所需要信息同样包括 Receiver 和 Intent Filter，以及 Action 的条件。

动态注册方式与静态注册方式不同，因为在程序运行中注册，所以当应用程序关闭后，Receiver 就不再进行监听。

下面这个例子通过按钮事件，在 Java 程序中动态实现 BroadcastReceiver 的注册和注销，见图 5.3。



图 5.3　动态实现 BroadcastReceiver 的注册和注销界面

要实现这个例子的功能，需要做下面工作：

(1) 定义具有三个按钮的用户界面的布局文件 main.xml 和相关的资源文件，代码略。

(2) 定义用户界面的 Activity，导入布局文件，并根据按钮不同的功能，在按钮单击事件处理代码中实现不同的功能，见代码 5.15。

(3) 定义 BroadcastReceiver，处理广播消息，见代码 5.16。

(4) 在 AndroidManifest.xml 中注册所定义的 Activity 和 BroadcastReceiver。

代码 5.15　TestDynamicRecieverActivity.java

```
import android.app.Activity;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class TestDynamicRecieverActivity extends Activity{
    //定义 Action 常量
    protected static final String ACTION="com.android.broadcast.RECEIVER_
    ACTION";
    private Button btnBroadcast;
    private Button registerReceiver;
    private Button unregisterReceiver;
    private MyReceiver receiver;
    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //创建 Intent 发送广播消息
        btnBroadcast=(Button)findViewById(R.id.btnBroadcast);
        btnBroadcast.setOnClickListener(new OnClickListener(){
```

```
            @Override
            public void onClick(View v){
                Intent intent=new Intent();
                intent.setAction(ACTION);
                sendBroadcast(intent);
            }
        });

        //动态注册 BroadcastReceiver
        registerReceiver=(Button)findViewById(R.id.btnregisterReceiver);
        registerReceiver.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View v){
                receiver=new MyReceiver();
                IntentFilter filter=new IntentFilter();
                filter.addAction(ACTION);
                registerReceiver(receiver, filter);
            }
        });

        //动态注销 BroadcastReceiver
        unregisterReceiver=(Button)findViewById(R.id.btnunregisterReceiver);
        unregisterReceiver.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View v){
                //注销 BroadcastReceiver
                unregisterReceiver(receiver);
            }
        });
    }
}
```

代码 5.16　MyReceiver.java

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class MyReceiver extends BroadcastReceiver{
    //定义日志标签
    private static final String TAG="Test";
    @Override
```

```
    public void onReceive(Context context, Intent intent){
        //输出日志信息
        Log.i(TAG, "MyReceiver onReceive--->");
    }
}
```

Java 程序代码编写完成后，不要忘记还需要在 AndroidMenifest.xml 文件中注册才能够运行。执行前面的代码，出现用户界面后，我们可以分步测试，观察结果日志，了解动态注册对 Receiver 功能的影响。

(1) 首先单击“发送广播”按钮的时候，因为程序没有注册 BraodcastReceiver，所以 MyReceiver 不会监听处理任何广播信息，LogCat 没有输出任何信息。

(2) 单击“注册广播接收器”按钮，程序会执行此按钮事件处理代码，动态地注册 BraodcastReceiver；再单击“发送广播”按钮，MyReceiver 会监听系统中的广播 Intent，并检测是否与注册的过滤条件匹配，这里发送的 Intent 的 Action 字段的值与动态注册的 Intent Filter 条件相同，系统会调用其 onReceive()方法处理这个广播消息，则 LogCat 会增添新的日志信息。

(3) 单击“注销广播监听器”按钮，程序会执行此按钮事件处理代码，动态地注销 BraodcastReceiver，MyReceiver 恢复到没有注册时的情况；再单击“发送广播”按钮，LogCat 没有输出任何信息。

5.3　Notification 管理

Notification，即通知，是一种消息，可以在应用程序界面显示其图形标记，提示用户。例如当用户操作应用时，如果有电话、短信或者邮件到达，可以向系统提交一个 Notification，它会首先以图标的形式显示在设备的状态栏位置，在手机的状态栏上就会出现一个小图标，提示用户处理这个消息，如图 5.4 所示。用户手从上方滑动状态栏就可以展开查看通知的详细信息，并进行处理。

通知的详细信息展开后的显示元素如图 5.5 所示。



图 5.4　提示图标

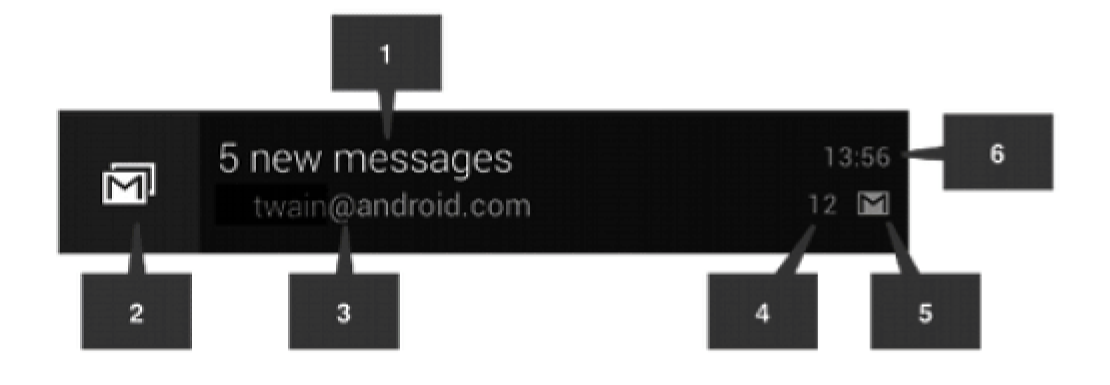


图 5.5　Notification 详细信息

对应图中的标号，元素分别为：

- 1：通知的标题；
- 2：大图标；
- 3：通知的内容；

- 4：通知的信息；
- 5：小图标；
- 6：通知发送的时间。

Android 提供了两种类型的 Notification 视图：正常视图（Normal View）和大视图（Big View），图 5.5 是正常的视图。还有一种是大视图，除了正常视图的元素之外，还包括消息的细节内容。消息细节内容的图形显示样式可以设置为不同的样式，包括大图样式、大文本样式、收件箱样式等。这些样式上还包括一些正常视图上没有的界面元素。

5.3.1 创建 Notification

Notification 与 Toast 都可以起到通知、提醒的作用，都可以随时取消。但它们的实现原理和表现形式却完全不一样。Toast 相当于一个定时关闭的对话框，是用户界面某个 Activity 的一部分，也可以用弹出的方式对用户的某个操作给出简单的反馈。Notification 显示在屏幕上方的状态栏中，在用户界面之外，也可以有闪烁、声音、震动等其他形式，是相对独立的。更重要的是，使用和查看 Notification 通常会对系统的任务栈产生影响，需要用 NotificationManager 来管理，而对于 Toast，只需要简单地创建对象并显示，不会对系统产生影响。

Android 系统的 NotificationCompat. Builder 类用于创建 Notification 对象。在应用需要时，可以使用 Notification. Builder. build()方法来创建一个 Notification 对象，并同时设置其界面显示图标和动作。

NotificationManager 类用于管理 Notification 对象。使用 NotificationManager. notify()可以将创建后的 Notification 对象向系统发布出去，在状态栏中显示出来。

在一个 Notification 对象中，小图标、标题和文本内容三个字段是必须赋值的。

在创建 Notification 对象时，可以由 Notification. Builder. build()方法直接给这三个字段赋值，也可以在创建完成后使用 setSmallIcon()方法、setContentTitle()方法和 setContentText()方法分别进行设置。其他的字段都是可选的。具体字段的细节，可以查阅 Notification. Builder 类的说明。

虽然其他字段都是可选的，但一般来说，每个 Notification 至少还是会设置一个动作 Action。例如用户在单击或触发 Notification 时，可以打开一个 Activity，或者关闭闹钟，或者打开一个浏览器等。一个 Notification 可以定义多个动作，在其被单击时同时触发。

Notification 对象中所设定的动作，是通过 PendingIntent 对象来定义的。PendingIntent 对象中包含一个 Intent，可以用来启动一个 Activity。通过 Notification. Builder 中的 setContentIntent()方法来将一个 PendingIntent 对象与一种操作关联起来。

创建和发布一个简单的 Notification 分下面几个步骤。

（1）创建一个 Notification 对象。

在创建一个 Notification 对象时，需要通过 Notification. Builder 定义 Notification 定义它的小图标、标题和文本内容。代码如下：

```
NotificationCompat.Builder mBuilder=
    new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("My notification")
    .setContentText("Hello World!");
```

(2) 定义 Notification 的 Action 动作。

一般来说，每个 Notification 至少会设置一个动作 Action，通过这个 Action 启动其他的 Activity，让用户进入应用程序的另一个用户界面，查看引起这个 Notification 的事件或做进一步处理。

Notification 的动作定义通过 PendingIntent 对象完成。具体代码实现时首先创建一个 Intent 对象，设置在 Notification 操作时要启动的 Activity；然后创建一个这个 Intent 对象的 PendingIntent 对象，具体如何构建 PendingIntent 对象与启动的 Activity 类型有关。

下面是简单的代码：

```
Intent resultIntent=new Intent(this, ResultActivity.class);

PendingIntent resultPendingIntent=
    PendingIntent.getActivity(
    this,
    0,
    resultIntent,
    PendingIntent.FLAG_UPDATE_CURRENT
);
```

(3) 设置 Notification 的单击行为。

如果要把上面定义的 PendingIntent 对象与一个用户操作相关联，需要调用 NotificationCompat.Builder 的对应方法。例如，当用户单击通知 Notification 的文本时，要启动一个 Activity，则通过 setContentIntent()方法添加前面定义的 PendingIntent 对象。代码如下：

```
mBuilder.setContentIntent(resultPendingIntent);
```

简单地说，PendingIntent 就是在 Intent 上加了指定的动作。对于 Intent 来说，只有在执行 startActivity()、startService()或 sendBroadcast()方法后，才能使 Intent 有用；而对于 PendingIntent 来说，本身就包含了这些方法的功能，还可以使用 PendingIntent.getActivity()和 PendingIntent.getService()方法来调用活动和服务。例如：

```
PendingIntent pi=PendingIntent.getActivity(this, 0, new Intent(this,
    HandleNotificationActivity.class), 0)
```

PendingIntent 还有 PendingIntent. getBroadcast()方法，其包含了 sendBroadcast()的功能

(4) 发布一个 Notification。

发布 Notification 时，首先要获取一个 NotificationManager 实例，使用 notify()方法发布 Notification 对象，然后使用 build()返回一个 Notification 对象。

```
//Sets an ID for the notification
int mNotificationId=001;
//Gets an instance of the NotificationManager service
NotificationManager mNotifyMgr=
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
//Builds the notification and issues it
mNotifyMgr.notify(mNotificationId, mBuilder.build());
```

到此为止，创建一个简单的 Notification 的代码就完成了。

下面通过一个简单的例子，说明如何在应用程序中创建和使用 Notification。在这个例子中，主界面是一个 Button，单击 Button 后创建一个简单的 Notification，这个 Notification 的具体内容由另一个 Activity 显示。

首先，在 SampleofAndroid 项目中定义主界面的布局文件，说明主界面中的 Button 单击后的事件处理代码由 createNotification()方法实现，见代码 5.17。

代码 5.17　simple_notif_layout. xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:onClick="createNotification"
        android:text="Create Notification">
    </Button>

</LinearLayout>
```

然后，定义 Notification 内容的查看界面的布局文件，见代码 5.18。

代码 5.18　simple_notif_result_layout. xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is the result activity opened from the
        notification">
    </TextView>

</LinearLayout>
```

创建展示 Notification 内容的 Activity，见代码 5.19。

代码 5.19 NotificationResultActivity.java

```
import android.app.Activity;
import android.os.Bundle;

public class NotificationResultActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. simple_notif_result_layout);
    }
}
```

创建主界面的 Activity，在其中实现 Button 单击事件处理器 createNotification()方法，在其中创建 Notification，见代码 5.20。

代码 5.20 CreateNotificationActivity.java

```
import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class CreateNotificationActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.main);
    }

public void createNotification(View view) {
    //Prepare intent which is triggered if the
    //notification is selected
    Intent intent=new Intent(this, NotificationReceiverActivity.class);
    PendingIntent pIntent=PendingIntent.getActivity(this, 0, intent, 0);

    //Build notification
    //Actions are just fake
    Notification noti=new Notification.Builder(this)
        .setContentTitle("New mail from "+"test@gmail.com")
        .setContentText("Subject").setSmallIcon(R.drawable.icon)
        .setContentIntent(pIntent)
        .addAction(R.drawable.icon, "Call", pIntent)
        .addAction(R.drawable.icon, "More", pIntent)
        .addAction(R.drawable.icon, "And more", pIntent).build();
      NotificationManager  notificationManager  =  (NotificationManager)
      getSystemService(NOTIFICATION_SERVICE);
    //Hide the notification after its selected
    noti.flags |=Notification.FLAG_AUTO_CANCEL;

    notificationManager.notify(0, noti);

  }
}
```

上面的 Notification 的例子是最简单的 Notification 格式，如果要向通知添加声音、闪灯和振动效果，最简单、最一致的方式是使用当前的用户默认设置，使用 Notification 的 defaults 属性。这些属性可以组合使用。例如：

```
Notification.DEFAULT_LIGHT
Notification.DEFAULT_SOUND
Notification.DEFAULT_VIBRATE
```

如果想全部使用默认值，可以使用 Notification. DEFAULT_ALL 常量。

```
notification.tickerText="hello";
```

通过向 sound 属性分配一个位置 URI，Android 可以将手机上的任意音频文件作为通知进行播放。

下面给出一些 Notification 提示方式的不同设置的例子。

(1) 在状态栏(Status Bar)显示的通知文本提示,代码如下:

```
notification.tickerText="hello";
```

(2) 发出提示音,代码如下:

```
notification.defaults=Notification.DEFAULT_SOUND;
notification.sound=Uri.parse("file:///sample/notification/ringer.mp3");
notification.sound= Uri.withAppendedPath(Audio.Media.INTERNAL_CONTENT_
URI, "6");
```

(3) 手机振动,代码如下:

```
notification.defaults=Notification.DEFAULT_VIBRATE;
long[] vibrate={0,100,200,300};
notification.vibrate=vibrate;
```

(4) LED 灯闪烁,代码如下:

```
    notification.defaults=Notification.DEFAULT_LIGHTS;
    //或者可以自己的 LED 提醒模式
    notification.ledARGB=0xff00ff00;
    notification.ledOnMS=300;          //亮的时间
    notification.ledOffMS=1000;        //灭的时间
    notification.flags=Notification.FLAG_SHOW_LIGHTS;
```

5.3.2 导航设计

应用程序创建完 Notification 后,Notification 会在某个条件满足时被发布,这时在用户界面上方的状态栏中会出现一个图标。例如,用户在玩游戏的时候收到了一封 Email,邮件到达的 Notification 就会显示在状态栏中。用户可以在任意时候滑动图标打开 Notification,查看其内容。

当用户从某个应用程序界面去查看一个 Notification 时,需要离开当前的 Activity,离开当前的应用程序流程,由 Notification 的 PendingIntent 去启动另一个 Activity,来展示 Notification 的内容。这个 Activity 可能是另一个应用程序正常流程中的一个界面,也可能仅仅是一个独立的显示界面。如果不做任何设置,当用户浏览完 Notification 按 Backspace 键时,无论前面的那种情况,用户界面会回到查看 Notification 之前的界面。如果希望针对这两种情况做不同的处理,在设计 Notification 时可以根据情况采用不同的 Activity 启动方式。

Notification 有两种常用的 Activity 启动方式:有规律的 Activity 和特定的 Activity。

如果 Notification 启动的 Activity 是一个应用程序正常流程的一部分,则这个需启动的 Activity 归为有规律的 Activity。在这种情况下,可以创建一个 PendingIntent 来启动一个新任务,即启动一个新的应用程序,并且给 PendingIntent 建立一个回退栈,这个回退栈中复制了这个程序流程中按 Backspace 键程序做出的正常反应行为。

例如，Gmail 邮件的 Notification 示范了这类固定的 Activity，当用户单击一个邮件的 Notification 时，用户可以看到信息；而使用 Backspace 键后，用户会回到 Gmail 收件箱的 Activiy。如果用户继续使用 Backspace 键，则回到 Home 界面上，这就像从 Home 界面进入 Gmail 应用效果一样。

这种情况的设置，通过 Notification 进入一个应用程序后，会按照这个应用程序的流程正常运行，与用户直接运行这个应用程序功能相同。如果用户持续按 Backspace 键或直接按 Home 键，最后会回到 Home 界面上，而不是最初查看 Notification 的应用程序界面。

如果从 Notification 启动的 Activity 并不是某个应用程序流程的一部分，而是为了显示 Notification 很难显示的信息或更细节的信息而创建的用户界面，是 Notification 的一个扩展，则将这个 Activity 定义为特定的 Activity。这种情况下，不需要创建回退栈，如果使用 Backspace 键，会直接回到 Home 界面。

(1) 为有规律的 Activity 创建 PendingIntent。

首先，在 Manifest 文件中定义应用程序的 Activity 层次结构。

对于 Android 4.0.3 和之前的版本，通过在<activity>元素中增加<meta-data>子元素来指定 Activity 的父级。在这个子元素中，设置 android:name="android.support.PARENT_ACTIVITY"和 android:value="<parent_activity_name>"，其中<parent_activity_name>是指父<activity>元素的名称。

对于 Android 4.1 和之后的版本，通过为<activity>元素增加 android:parentActivityName 属性来设置，见代码 5.21。

代码 5.21　Manifest 中定义 Activity 层次结构

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name=".ResultActivity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

其次，基于启动 Activity 的 Intent 创建回退栈。创建回退栈需要完成下面几个设置：

① 创建启动 Activity 的 Intent。

② 调用 TaskStackBuilder.create()创建栈创建器。

③ 调用 addParentStack()把回退栈加入栈中，对于定义在 manifest 层次中的每一个 Activity，回退栈中都包含一个启动其的 Intent 对象。

④ 调用 addNextIntent()添加从 Notification 中启动 Activity 的 Intent，把前面所创建的 Intent 作为参数。

⑤ 如果需要给栈中的 Intent 对象添加参数，可以调用 TaskStackBuilder.editIntentAt()，这可以保证让用户在使用 Backspace 键导航时，确保目标 Activity 显示有效的数据。

⑥ 调用 calling getPendingIntent()方法从回退栈中获取一个 PendingIntent，然后把这个 PendingIntent 作为 setContentIntent()方法的参数。

具体的设置过程可以参考代码 5.22。

代码 5.22　创建回退栈

```
…
Intent resultIntent=new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder=TaskStackBuilder.create(this);
//Adds the back stack
stackBuilder.addParentStack(ResultActivity.class);
//Adds the Intent to the top of the stack
stackBuilder.addNextIntent(resultIntent);
//Gets a PendingIntent containing the entire back stack
PendingIntent resultPendingIntent=
    stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
…
NotificationCompat.Builder builder=new NotificationCompat.Builder(this);
builder.setContentIntent(resultPendingIntent);
NotificationManager mNotificationManager=
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.notify(id, builder.build());
```

(2) 使用特定的 Activity。

如果使用特定的 Activity，就不需要使用回退栈，也不需要在 Manifest 中定义 Activity 的层次结构等，而是需要设置 Activity 的任务选项，并且使用 getActivity()方法创建 PendingIntent。

首先在 Manifest 文件中添加"android:taskAffinity"和"android:excludeFromRecents"这两个<activity>的属性的设置，见代码 5.23。

代码 5.23　Manifest 说明特定的 Activity 属性

```
<activity
    android:name=".ResultActivity"
…
    android:launchMode="singleTask"
    android:taskAffinity=""
```

```
    android:excludeFromRecents="true">
</activity>
...
```

其次，建立和发布 Notification，见代码 5.24。

① 创建启动 Activity 的 Intent。

②通过调用 setFlags()设置新的空任务，把 FLAG_ACTIVITY_NEW_TASK 和 FLAG_ACTIVITY_CLEAR_TASK 作为标记参数。

③ 设置 Intent 的其他需要的选项。

④ 通过调用 getActivity()方法使用 Intent 创建一个 PendingIntent 对象，然后把这个 PendingIntent 作为 setContentIntent()方法的参数。

代码 5.24　建立和发布 Notification

```
//Instantiate a Builder object.
NotificationCompat.Builder builder=new NotificationCompat.Builder(this);
//Creates an Intent for the Activity
Intent notifyIntent=
        new Intent(new ComponentName(this, ResultActivity.class));
//Sets the Activity to start in a new, empty task
notifyIntent.setFlags(FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK);
//Creates the PendingIntent
PendingIntent notifyIntent=
        PendingIntent.getActivity(
        this,
        0,
        notifyIntent
        PendingIntent.FLAG_UPDATE_CURRENT
);

//Puts the PendingIntent into the notification builder
builder.setContentIntent(notifyIntent);
//Notifications are issued by sending them to the
//NotificationManager system service.
NotificationManager mNotificationManager=
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
//Builds an anonymous Notification object from the builder, and
//passes it to the NotificationManager
mNotificationManager.notify(id, builder.build());
```

Notification 通过有规律的 Activity 启动方式和特定的 Activity 启动方式，都可以使用户在查看 Notification 之后，通过 Backspace 键和 Home 键回到 Home 界面。如果创建 Notification 时不做任何设置，如同 5.3.1 的例子一样，则 Backspace 键会使用户界面回到查看 Notification 之前的界面。

5.3.3　定义样式

Android 4.1 对 Android 的 Notification 框架进行了重大的更新。应用程序现在可以通过单击选择展开或者折叠来显示更大、更丰富的 Notification。Notification 支持包括照片在内的新内容类型，支持优先级的配置，以及多个动作的设置。使用改进的 Notification，应用程序可以创建的使用面积较大，高达 256dp 的高度的 Notification 信息。

Android 系统提供的 Notification 主要包括以下四种类型。

- 基本类型，其使用图标显示简单的、短的通知信息。
- 大图片类型(Big Picture Style)，其可以显示图片的内容，例如位图。
- 大文本类型(Big Text Style)，其可以显示多个 TextView。
- 收件箱类型(Inbox Style)，其可以显示任何类型的列表。

我们前面的例子都是创建的基本类型的 Notification。基本类型 Notification 呈现的是正常视图(Normal View)，而后面三种类型的 Notification 都属于大视图(Big View)模式。下面简单介绍大视图模式的 Notification 如何创建。

在 Android 4.1 之前的版本中，需要程序员直接创建 Notification 对象，而 Android 4.1 之后的版本可以使用 Notification. Builder 类来创建 Notification 对象，简化了 Notification 对象创建的过程，而且可以根据需求，使用 Notification. BigPictureStyle、Notification. BigTextStyle 和 Notification. InboxStyle 工具类，创建各种类型的通知。下面是如何创建大视图模式 Notification 的三种样式的代码示范。

(1) 大文本类型。

```
Builder builder=new Notification.Builder(this);
builder.setContentTitle("Big text Notofication")
    .setContentText("Big text Notification")
    .setSmallIcon(R.drawable.ic_launcher)
    .setAutoCancel(true);
    .setPriority(Notification.PRIORITY_HIGH)
    .addAction(R.drawable.ic_launcher_web, "show activity", pi);
Notification notification=new Notification.BigTextStyle(builder)
    .bigText(msgText).build()
```

(2) 大图片类型。

```
Builder builder=new Notification.Builder(this);
  builder.setContentTitle("BP notification")
    //Notification title
    .setContentText("BigPicutre notification")
    //you can put subject line.
    .setSmallIcon(R.drawable.ic_launcher)
    //Set your notification icon here.
    .addAction(R.drawable.ic_launcher_web, "show activity", pi)
```

```
    .addAction(
      R.drawable.ic_launcher_share,
      "Share",
      PendingIntent.getActivity(getApplicationContext(), 0,
        getIntent(), 0, null));

//Now create the Big picture notification.
Notification notification=new Notification.BigPictureStyle(builder)
    .bigPicture(
      BitmapFactory.decodeResource(getResources(),
        R.drawable.big_picture)).build();
//Put the auto cancel notification flag
notification.flags |=Notification.FLAG_AUTO_CANCEL;
```

（3）收件箱类型。

```
Builder builder=new Notification.Builder(this)
    .setContentTitle("IS Notification")
    .setContentText("Inbox Style notification!!")
    .setSmallIcon(R.drawable.ic_launcher)
    .addAction(R.drawable.ic_launcher_web, "show activity", pi);

Notification notification=new Notification.InboxStyle(builder)
    .addLine("First message").addLine("Second message")
    .addLine("Thrid message").addLine("Fourth Message")
    .setSummaryText("+2 more").build();
//Put the auto cancel notification flag
notification.flags |=Notification.FLAG_AUTO_CANCEL;
```

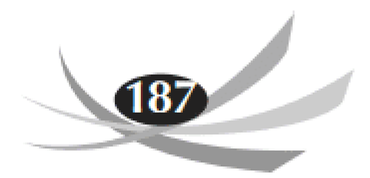
在 Notification 的样式设置完成后，创建 Notification 的其余步骤与基本样式类似。

5.4 本 章 小 结

本章主要介绍了 Android 系统中用于消息传递的组件 Intent、BroadcastReciever 和 Notification。

Intent 是 Android 的一个基础组件。通过 Intent 的消息创建、消息触发、消息传递和消息响应，Android 系统实现窗口跳转、传递数据或调用外部程序，进行应用程序的激活和调用。Android 的三大基础组件 Activity、Service、BroadcastReceiver，都可以通过定义 Intent 的消息，实现在各组件之间的程序跳转和数据传递。Intent 对象中同时携带触发其他组件执行的条件信息和触发后该组件执行时所需要的信息。

使用 Intent 来实现程序跳转和数据传递有两种不同方式：显式和隐式。在 Intent 对象的组件名称属性 Component Name 里直接设定要激活的组件，这种方式称为显式；



Activity、Service、BroadcastReceiver 可以在 Manifest 文件中，通过 Intent 过滤器设置一个或多个条件，说明该组件可接收的 Intent 对象，过滤掉不想接收的隐式 Intent 对象，这种方式称为隐式。

BroadcastReceiver 是 Android 系统中负责接收广播消息并对消息做出反应的组件。BroadcastReceiver 没有用户界面，可以接收到信息后启动 Activity 或者通过 NotificationManger 通知用户，也可以通过其他多种方式通知用户。多数的广播是系统发起，如果是用户应用程序发送广播消息，在 Intent 对象创建后，启动 BroadcastRecevicer 的方式有两种：sendBroadcast()方法和 sendOrderedBroadcast()方法。

Notification，即通知，是一种消息，以图标的形式显示在设备的状态栏位置，提示用户。Android 提供了两种类型的 Notification 视图：正常视图(Normal View)和大视图(Big View)。Android 系统的 NotificationCompat. Builder 类用于创建 Notification 对象。在应用需要时，可以使用 Notification. Builder. build()方法来创建一个 Notification 对象，并同时设置其界面显示图标和动作。

第 6 章

多任务与服务

6.1 基本概念

在 Android 系统中，如果有一个应用程序组件是第一次被启动，而且这时候应用程序也没有其他组件在运行，则 Android 系统会为应用程序创建一个 Linux 进程，这个 Linux 进程只包含一个线程。举个例子，如果一个应用程序启动了第一个 Activity，这个 Activity 里有一个文本框和一个按钮，这时，Android 系统会为应用程序创建一个单线程的 Linux 进程，初始化这个文本框和按钮，当这个应用程序启动另一个 Activity 时，初始化图形组件的还是这个已经创建好的线程，不会再创建新的。也就是说，这个应用程序会一直单线程单任务运行图形组件的初始化和与图形组件相关的操作。

默认情况下，同一个应用程序的所有组件都运行在同一个进程和线程里，这个线程叫做主线程。如果一个组件启动时，应用程序的其他组件已经在运行了，则此组件会在已有的进程和线程中启动运行。

如果希望 Android 应用程序实现多任务，可以通过代码指定组件运行在其他进程里，或为进程创建额外的线程。

下面介绍 Android 的进程调度机制。

6.1.1 进程

默认情况下，同一个应用程序内的所有组件都是运行在同一个进程中的，大部分应用程序都是按照这种方式运行的。但在具体应用中，很多时候需要通过在 Manifest 文件中进行设置，指定某个特定组件归属于哪个进程。

可以通过 manifest. xml 文件设定应用程序归属的进程。Manifest 文件中的每种组件元素——<activity>、<service>、<receiver>和<provider>——都支持定义 android:process 属性，用于指定组件运行的进程。

设置这个属性就可实现每个组件在各自的进程中运行，或者某几个组件共享一个进程而其他组件运行于独立的进程。设置这个属性也可以让不同应用程序的组件运行在同一个进程中，这就实现了多个应用程序共享同一个 Linux 用户 ID、赋予同样的权限。

<application>元素也支持 android:process 属性，用于指定所有组件的默认进程。

Android 一个重要并且特殊的特性就是，一个应用的进程的生命周期不是由应用程序自身直接控制的，而是由系统根据运行中的应用的一些特征来决定的，包括这些应用程序对用户的重要性、系统的全部可用内存。

大部分情况下，每个 Android 应用程序都将运行在自己的 Linux 进程当中。当这个应用的某些代码需要执行时，进程就会被创建，并且将保持运行，直到该进程不再需要，而系统需要释放它所占用的内存，为其他应用所用时才停止。

Android 系统试图尽可能长时间地保持应用程序进程，但为了新建或者运行更加重要的进程，总是需要清除过时进程来回收内存。为了决定保留或终止哪个进程，根据进程内运行的组件及这些组件的状态，系统把每个进程都划入一个“重要性层次结构”中。重要性最低的进程首先会被清除，然后是下一个最低的，依此类推，这都是回收系统资源所必需的。

重要性层次结构共有五级，按照重要程度列出了各类进程：前台进程、可见进程、服务进程、后台进程和空进程，其中第一类进程是最重要的，将最后一个被终止。

1. 前台进程

用户当前操作所必须的进程。满足以下任一条件时，进程被视作处于前台：

- 正在与用户交互的 Activity 进程(例如 Activity 的 onResume()方法已被调用)。
- 正在与用户交互的 Activity 绑定的 Service 进程。
- 正在运行前台 Service 进程，例如 Service 被 startForeground()方法调用。
- 正在运行生命周期回调方法(例如 onCreate()、onStart()或 onDestroy())的 Service。
- 正在运行 onReceive()方法的 BroadcastReceiver。

一般而言，任何时刻只有很少的前台进程同时运行。只有当内存不足以维持它们同时运行时，作为最后的策略它们才会被终止。通常，终止一些前台进程是为了保证用户界面的及时响应。

2. 可见进程

如果进程没有任何前台组件但仍会影响用户在屏幕上所见内容的进程，称为可见进程。满足以下任一条件时，进程被认为是可见的：

- 如果 Activity 不在前台，但用户仍然可见(例如 Activity 的 onPause()方法被调用了)。如当前台 Activity 打开了一个对话框，而之前的 Activity 还允许显示在后面，但是已经无法与用户进行交互了。
- 一个绑定到可见或前台 Activity 的 Service 进程。

可见进程被认为是非常重要的，除非无法维持所有前台进程同时运行了，否则它们是不会被终止的。

3. 服务进程

对于由 startService()方法启动的 Service 进程，它不会升级为上述两种级别。尽管服务进程不直接和用户所见内容关联，但它们通常在执行一些用户关心的操作，如在后台播放音乐或从网络下载数据等。因此，除非内存不足以维持所有前台、可见进程同时运行，系统会保持服务进程的运行。

4. 后台进程

后台进程包含目前用户不可见 Activity(例如 Activity 的 onStop()方法已被调用)的进程。这些进程对用户体验没有直接的影响，系统可能在任意时间终止它们，以回收内存

供前台进程、可见进程及服务进程使用。通常会有很多后台进程在运行，所以它们被保存在一个 LRU(最近最少使用)列表中，以确保最近被用户使用的 Activity 最后一个被终止。如果一个 Activity 正确实现了生命周期方法，并保存了当前的状态，则终止此类进程不会对用户体验产生可见的影响。因为在用户返回时，Activity 会恢复所有可见的状态。如果要了解关于保存和恢复状态的详细信息，可以参阅 Activity 生命周期文档。

5. 空进程

空进程即不含任何活动应用程序组件的进程。保留这种进程的唯一目的就是用作缓存，以改善下次在此进程中运行组件的启动时间。为了在进程缓存和内核缓存间平衡系统整体资源，系统经常会终止这种进程。

依据进程中目前活跃组件的重要程度，Android 会给进程评估一个尽可能高的级别。例如：如果一个进程中运行着一个服务和一个用户可见的 Activity，则此进程会被评定为可见进程，而不是服务进程。

此外，一个进程的级别可能会由于其他进程的依赖而被提高，为其他进程提供服务的进程级别永远不会低于使用此服务的进程。

因为运行服务的进程级别是高于后台 Activity 进程的，所以，如果 Activity 需要启动一个长时间运行的操作，则为其启动一个服务 Service 会比简单地创建一个工作线程更好些，尤其是在此操作时间比 Activity 本身存在时间还要长久的情况下。

6.1.2 线程

应用程序启动时，系统会为它创建一个名为 main 的主线程。主线程非常重要，因为它负责把事件分发给相应的用户界面 widget——包括屏幕绘图事件。它也是应用程序与 Android UI 组件包(来自 android. widget 和 android. view 包)进行交互的线程。因此，主线程有时也被叫做 UI 线程。

系统并不会为每个组件的实例都创建单独的线程。运行于同一个进程中的所有组件都是在 UI 线程中实例化的，对每个组件的系统调用也都是由 UI 线程分发的。

如果应用程序在与用户交互的同时需要执行繁重的任务，用户单线程模式可能会导致运行性能很低下。例如，在查询数据库时，应用程序就需要做两件事，一是需要与数据库连接，访问数据库，获取查询结果；二是要初始化显示界面的组件，把获取的数据给显示出来。因为是单线程，就必须先做完第一件事后才能做第二件事。这个过程有可能因为网络状况或数据库繁忙，在访问数据库、获取结果数据时花费比较长的时间，导致不能执行用户显示界面的初始化，使得用户界面呈现出静止状态。这种状态，称为 UI 线程阻塞。如果 UI 线程被阻塞超过一定时间(目前大约是 5 秒钟)，就会弹出对话框，提示用户应用程序没有响应(ANR)。图 6.1 就是这个对话框。

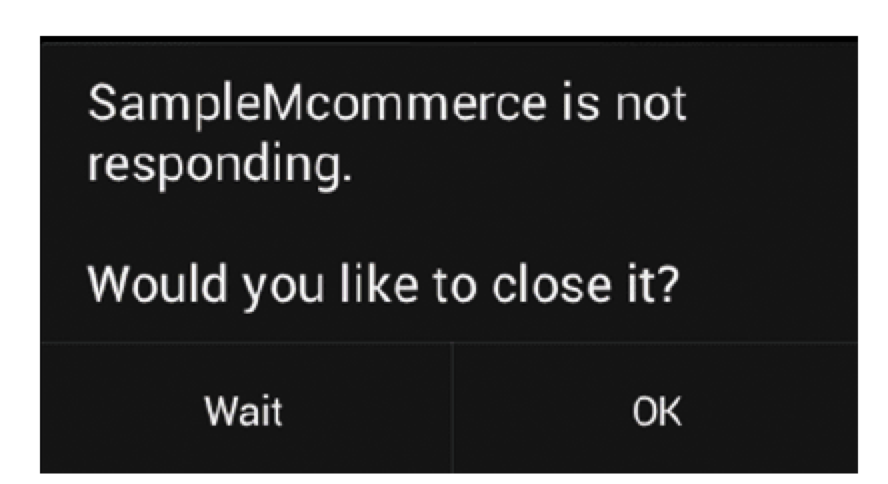
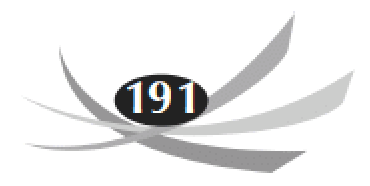


图 6.1 UI 线程阻塞提示对话框

Android 的单线程模式遵守两个规则：不要阻塞 UI 线程和不要在 UI 线程之外访问 Android 的 UI 组件包。

这样，程序才能有友好的界面，顺利运行。



一般稍微复杂一点的应用程序，特别是需要网络访问或数据库访问的应用程序，都需要使用多任务的方式。

在 Android 应用程序中，我们创建的 Activity、Service、Broadcast 等都是在主线程(UI 线程)处理的，但一些比较耗时的操作，如 I/O 读写的大文件读写，进行数据库操作以及网络下载需要很长时间，为了不阻塞用户界面，会出现响应提示窗口，这时我们可以考虑创建一个工作线程(继承 Thread 类或者实现 Runnable 接口)来解决。

6.2　实现多任务

Android 多任务的调度和实现采用消息驱动机制。熟悉 Windows 编程的朋友可能知道 Windows 程序是消息驱动的，并且有全局的消息循环系统。而 Android 应用程序也是消息驱动的，Google 参考了 Windows 系统，也在 Android 系统中实现了消息循环机制。Android 通过 Looper、Handler、MessageQueue 和 Message 来实现消息循环机制，Android 消息循环是针对线程的，即主线程和工作线程都可以有自己的消息队列和消息循环。

6.2.1　多任务实现原理

对于多线程的 Android 应用程序来说，有两类线程：一类是主线程，也就是 UI 线程；另一类是工作线程，也就是主线程或工作线程所创建的线程。Android 的线程间消息处理机制主要是用来处理主线程跟工作线程间通信的。图 6.2 是线程间通信原理图。

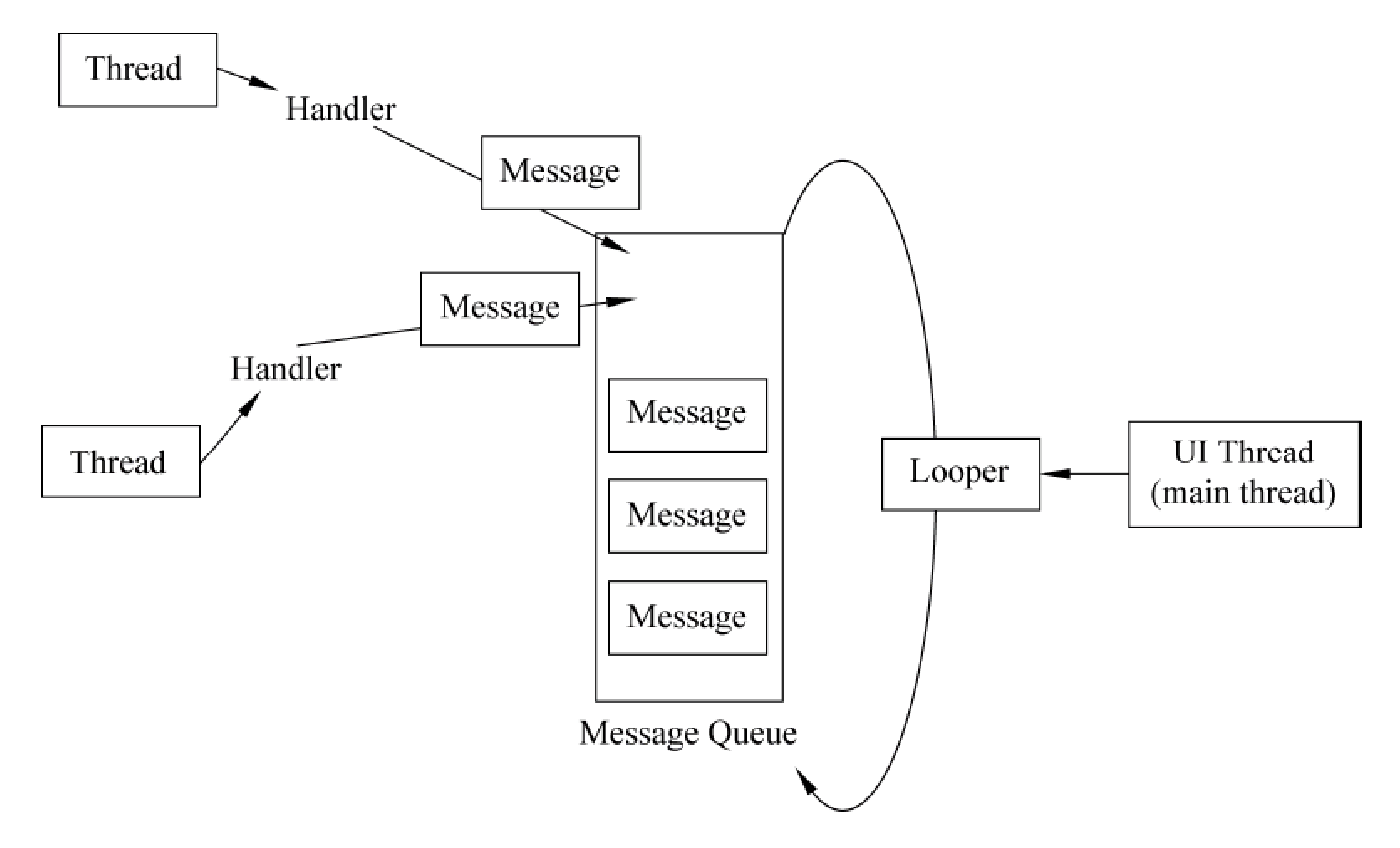


图 6.2　线程间通信原理

Android 应用程序是通过消息来驱动的，即在应用程序的主线程(UI 线程)中有一个消息循环，负责处理消息队列中的消息。

例如，当从网上下载文件时，为了不使主线程被阻塞，通常需要创建一个子线程来负责下载任务，同时，在下载的过程中，将下载进度以百分比的形式在应用程序的界面上显

示出来，这样既不会阻塞主线程的运行，又能获得良好的用户体验。但是，Android 应用程序的子线程是不可以操作主线程的 UI 的，那么，这个负责下载任务的子线程应该如何在应用程序界面上显示下载的进度呢？如果我们能够在子线程中往主线程的消息队列中发送消息，那么问题就迎刃而解了，因为发往主线程消息队列的消息最终是由主线程来处理的，在处理这个消息时，就可以在应用程序界面上显示下载进度了。

线程之间和进程之间是不能直接传递消息的，必须通过对消息队列和消息循环的操作来完成。Android 消息循环是针对线程的，每个线程都可以有自己的消息队列和消息循环。Andriod 提供了 Handler 类和 Looper 类来访问消息队列(Message Queue)。

Looper 类是用来封装消息循环和消息队列的一个类，负责管理线程的消息队列和消息循环，用于在 Android 线程中进行消息处理。Looper 对象是什么呢？其实 Android 中每一个线程都对应一个 Looper，Looper 可以帮助线程维护一个消息队列，是负责在多线程之间传递消息的一个循环器，线程通过 Looper 对象可以读写某个消息循环队列。使用 Looper.myLooper()得到当前线程的 Looper 对象，使用 Looper.getMainLooper()可以获得当前进程的主线程的 Looper 对象。

一个线程可以存在也可以不存在一个消息队列和一个消息循环，工作线程默认是没有消息循环和消息队列的。如果想让工作线程具有消息队列和消息循环，需要在线程中首先调用 Looper.prepare()来创建消息队列，然后调用 Looper.loop()进入消息循环，见代码 6.1。

代码 6.1 CustomThread.java

```
class CustomThread extends Thread {
      @Override
    public void run() {
        //建立消息循环的步骤
        //1. 初始化 Looper
        Looper.prepare();
        //2. 绑定 handler 到 CustomThread 实例的 Looper 对象
        mHandler=new Handler() {
        //3. 定义处理消息的方法
            public void handleMessage(Message msg) {
        switch(msg.what) {
                case MSG_HELLO:
                    Log.d("Test", "CustomThread receive msg:"
                            + (String) msg.obj);
                }
            }
        };
        //4. 启动消息循环
        Looper.loop();
    }
}
```

通过代码 6.1 的设置，工作线程 CustomThread 就具有了消息队列和消息循环的处理机制，可以在 Handler 中进行消息处理。代码中定义的 Handler 对象，其作用是把消息加入特定的消息队列中，并分发和处理该消息队列中的消息。

每个 Activity 是一个 UI 线程，运行于主线程中。Android 系统在启动的时候会为 Activity 创建一个消息队列和消息循环。

一个 Activity 中可以创建多个工作线程或者其他的组件，如果这些线程或者组件把它们的消息放入 Activity 的主线程消息队列，那么该消息就会在主线程中处理了。因为主线程一般负责界面的更新操作，并且 Android 系统中的界面控件都是单线程模式，多线程控制需要程序员实现，也就是非线程安全的，所以这种方式可以很好地实现 Android 界面更新。在 Android 系统中这种机制有广泛的应用。

那么一个工作线程怎样把消息放入主线程的消息队列呢？答案是通过 Handle 对象。只要 Handler 对象由主线程的 Looper 创建，那么调用 Handler 的 sendMessage()等方法，就会把消息放入主线程的消息队列；在主线程中调用 Handle 的 handleMessage()方法来处理消息，在这个方法中实现主线程的界面控件的操作，从而实现了工作线程和主线程之间的调度。

下面是一个简单的例子，在 Activity 中定义了 Handler 对象 h，并定义了一个工作进程 MyThread，在工作进程中使用对象 h 的 sendMessage()发送了一条消息到主线程的消息队列，见代码 6.2。

代码 6.2　MyHandler.java

```
public class MyHandler extends Activity {
    static final String TAG="Handler";
    static final int HANDLER_TEST=1;
    Handler h=new Handler() {
        public void handleMessage(Message msg) {
            switch(msg.what) {
            case HANDLER_TEST:
                Log.d(TAG, "The handler thread id="
                        +Thread.currentThread().getId()+" ");
                break;
            }
        }
    };

    /** Called when the activity is first created. * /
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "The main thread id="+Thread.currentThread().getId()
                +" ");
```

```
        new myThread().start();
        setContentView(R.layout.main);
    }

    class MyThread extends Thread {
        public void run() {
            Message msg=new Message();
            msg.what=HANDLER_TEST;
            h.sendMessage(msg);
            Log.d(TAG, "The worker thread id="
                    +Thread.currentThread().getId()+" ");
        }
    }
}
```

编译运行代码 6.2 后，可以看到有三条打印信息，显示了在这个 Activity 运行过程中，各个模块所处的线程情况。

在这个例子中，主线程在 onCreate()方法中通过 new myThread().start()启动了工作线程，工作线程 MyThread 中 run()方法中的执行代码，访问了主线程 Handler 对象 h，并在调用 Handler 的对象 h 时，向主线程消息队列加入了一条消息。这个过程中会不会出现消息队列数据不一致问题呢？因为 Handler 对象管理的 Looper 对象是线程安全的，不管是加入消息到消息队列和从队列读出消息都是有同步对象保护的，Handler 对象不会出问题。由于这里没有修改 Handler 对象，因此 Handler 对象不可能会出现数据不一致的问题。

工作线程和主线程运行在不同的线程中，所以必须要注意这两个线程间的竞争关系。在主线程中构造 Handler 对象，并且启动工作线程之后不要再修改，否则会出现数据不一致。这样在工作线程中可以放心地调用 SendMessage()等方法传递消息，Handler 对象的 handleMessage()方法将会在主线程中调用。在这个方法可以安全的调用主线程中任何变量和函数，进而完成更新 UI 的任务。

Android 有两种方式实现多线程操作 UI：第一种是创建新线程 Thread，用 Handler 负责线程间的通信和消息。第二种方式 AsyncTask 异步执行任务。

下面如何使用 Handle 实现多任务。

6.2.2 用 Handler 实现多任务

android.os.Handler 是 Android SDK 中处理定时操作的核心类。通过 Handler 类，可以提交和处理一个 Runnable 对象。这个对象的 run()方法可以立刻执行，也可以在指定时间之后执行(可以称为预约执行)。

Handler 类有两种主要用途：按照时间计划，在未来某时刻，对处理一个消息或执行某个 runnable 实例。把一个对另外线程对象的操作请求放入消息队列中，从而避免线程间冲突。

当一个进程启动时，主线程独立执行一个消息队列，该队列管理着应用顶层的对象（如 Activity、BroadcastReceiver 等）和所有创建的窗口。可以创建自己的一个线程，并通过 Handler 来与主线程进行通信。这可以通过在新的线程中调用主线程的 Handler 的 postXXX 和 SendMessage()方法来实现。

使用 post()方法实现多任务的主要步骤如下：

（1）创建一个 Handler 对象。

（2）将要执行的操作写在线程对象的 run()方法中。

（3）使用 post()方法运行线程对象。

（4）如果需要循环执行，需要在线程对象的 run()方法中再次调用 post()方法。

相关代码见代码 6.3。

代码 6.3　HandlerActivity. java

```
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.ProgressBar;
import android.widget.TextView;
import android.widget.Toast;

import com.taobao.mcommerce.sample.R;

public class HandlerActivity extends Activity implements OnClickListener {
    private static final String TAG="HandlerActivity";
    private Handler countHandler=new Handler();
    private TextView tvCount;
    private ProgressBar mProgressBar;
    private int count=0;

    private Runnable mRunToast=new Runnable() {
        @Override
        public void run() {
            Toast.makeText(HandlerActivity.this, "",
                    Toast.LENGTH_LONG).show();
        }
    };

    private Runnable mRunCount=new Runnable() {
```

```
        @Override
        public void run() {
            //TODO Auto-generated method stub
            tvCount.setText("Count: "+String.valueOf(++count));
            countHandler.postDelayed(this, 1000);
        }

    };
    private Handler updateProgressBarHandler=new Handler() {

        @Override
        public void handleMessage(Message msg) {
            mProgressBar.setProgress(msg.arg1);
            updateProgressBarHandler.post(mUpateProgressBarThread);
        }

    };
    private Runnable mUpateProgressBarThread=new Runnable() {

        int i=0;

        @Override
        public void run() {
            log("Begin Thread");
            i=i+10;
            Message msg=updateProgressBarHandler.obtainMessage();
            msg.arg1=i;
            try { Thread.sleep(1000);
            } catch(InterruptedException e) {
                //TODO Auto-generated catch block
                e.printStackTrace();
            }
    updateProgressBarHandler.sendMessage(msg);
            if(i==100) {
                updateProgressBarHandler
                        .removeCallbacks(mUpateProgressBarThread);
            }
        }
    };

    @Override
    public void onClick(View view) {
```

```
        switch(view.getId()) {
        case R.id.btnStart:
            countHandler.postDelayed(mRunCount, 1000);
            break;
        case R.id.btnStop:
            countHandler.removeCallbacks(mRunCount);
            break;
        case R.id.btnShowToast:
            countHandler.postAtTime(mRunToast,
            android.os.SystemClock.uptimeMillis()+15 * 1000);
            break;
        case R.id.btnUpdateProgressBar:
            mProgressBar.setVisibility(View.VISIBLE);
            updateProgressBarHandler.post(mUpateProgressBarThread);
            break;
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.c06_handler_layout);
        ((Button) findViewById(R.id.btnStart)).setOnClickListener(this);
        ((Button) findViewById(R.id.btnStop)).setOnClickListener(this);
        ((Button) findViewById(R.id.btnShowToast)).setOnClickListener
        (this);
        ((Button) findViewById(R.id.btnUpdateProgressBar))
                .setOnClickListener(this);
        mProgressBar=(ProgressBar) findViewById(R.id.progressBar1);
        tvCount=(TextView) findViewById(R.id.tvCount);
    }

    private void log(String msg) {
        Log.d(TAG, msg);
    }
}
```

6.2.3 AsyncTask 实现多任务

用 Handler 类来在子线程中更新 UI 线程虽然避免了在主线程进行耗时计算，但费时的任务操作总会启动一些匿名的子线程，太多的子线程给系统带来巨大的负担，随之带来一些性能问题。因此，Android 提供了一个工具类 AsyncTask 来实现异步执行任务。AsyncTask 类擅于处理一些后台比较耗时的任务，给用户带来良好的用户体验，不再需

要子线程和 Handler 就可以完成异步操作并且刷新用户界面。

如果要使用 AsyncTask，需要创建 AsyncTask 类，并实现其中的抽象方法以及重写某些方法。利用 AsyncTask 不需要自己来写后台线程，无须终结后台线程，但是 AsyncTask 的方式对循环调用的方式并不太合适。AsyncTask 是抽象类，AsyncTask 定义了三种泛型：Params、Progress 和 Result。它们的含义分别为：

- Params：表示启动任务执行的输入参数，如 HTTP 请求的 URL。
- Progress：表示后台任务执行的百分比。
- Result：表示后台执行任务最终返回的结果，如 String，Integer 等。

我们通过继承一个 AsyncTask 类来定义一个异步任务类。Android 提供一个让程序员编写后台操作更为容易和透明 AsyncTask，使得后台线程能够在 UI 主线程外进行处理。

AsyncTask 实现多任务，实现步骤如下：

(1) 使用 execute()方法触发异步任务的执行。

(2) 使用 onPreExecute()执行预处理，如绘制一个进度条控件。

(3) 使用 doInBackground()执行较为费时的操作，这个方法是 AsyncTask 的关键，必须覆盖重写。

(4) 使用 onProgressUpdate()对进度条控件根据进度值做出具体的响应。

(5) 使用 onPostExecute()对后台任务的结果做出处理。

相关代码见代码 6.4。

代码 6.4　AsyncTaskActivity.java

```
import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.ProgressBar;
import android.widget.TextView;

import com.taobao.mcommerce.sample.R;

public class AsyncTaskActivity extends Activity implements
OnClickListener {
    private Button Btn;
    private TextView txt;
    private int count=0;
    private boolean isRunning=false;
    private ProgressBar progressBar;

    /** Called when the activity is first created. */
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.c06_async_task_layout);
        Btn=(Button) findViewById(R.id.button1);
        txt=(TextView) findViewById(R.id.textView1);
        progressBar=(ProgressBar) findViewById(R.id.progressBar1);
        Btn.setOnClickListener(this);
    }

    public void onClick(View arg0) {
        //TODO Auto-generated method stub
        TimeTickLoad timetick=new TimeTickLoad();
        timetick.execute(1000);
    }

    private class TimeTickLoad extends AsyncTask<Integer, Integer, String>
{

        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            txt.setText("");
            progressBar.setVisibility(View.VISIBLE);
        }

        @Override
        protected String doInBackground(Integer… params) {
            for (int i=0; i<=10; i++) {
                publishProgress(i * 10);
                try {
                    Thread.sleep(params[0]);
                } catch(InterruptedException e) {
                    e.printStackTrace();
                }
            }
            return "";
        }

        @Override
        protected void onProgressUpdate(Integer… progress) {
            super.onProgressUpdate(progress);
```

```
            progressBar.setProgress(progress[0]);
        }

        @Override
        protected void onPostExecute(String result) {
            super.onPostExecute(result);
            txt.setText(result);
        }

    }
}
```

6.3 理解服务

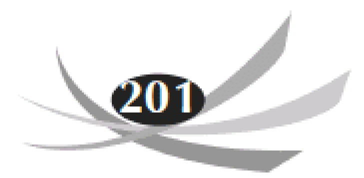
Service 是 Android 的四大组件之一，用于支持 Android 系统的服务。Service 是一个能够在后台执行长时间运行的操作应用程序组件，不提供用户界面。Android 的其他应用的组件可以在后台启动一个 Service 运行，即使用户切换到另一个应用，此 Service 也会继续运行。Android 服务组件就像是 Windows 系统服务或者 UNIX 的守护进程，这些都是在后台运行而不可见。

Service 不能与用户交互，也不能自己启动，需要调用 Context.startService()或 bindService()来启动，在后台运行。当应用程序需要进行某种不在前台显示的计算或数据处理时，就可以启动一个 Service 来完成，每个 Service 都继承自 android.app 包下的 Service 类。每个 Service 都必须在 AndroidManifest.xml 中通过<service>进行声明。

Service 具有自己的生命周期。Service 服务的生命周期是与 Activity 生命周期分离的，当 Activity 被暂停、停止或者销毁时，Service 组件还可以继续处理其他任务。例如，一个服务可以处理网络事务、播放音乐、执行文件 I/O 或者跟内容提供器交互，所有这些都是在后台完成的。

Android 支持服务有两个原因：一是允许我们方便地执行后台任务，二是实现同一设备上应用之间的跨进程通信。基于这两个原因，Android 系统支持两种类型服务，分别是本地服务和远程服务。本地服务是指只可以被驻留服务的应用访问的服务，而不能被本设备上的其他应用访问；远程服务既可以被其所驻留的应用访问，也可以被设备上的其他应用访问。例如在开发一个邮件应用时，可以创建一个本地服务实现邮件的发送，这是由于邮件的发送需要网络连接，这是一个耗时操作，需要在后台执行；另外一种情况，如果一个设备上很多程序都需要一个通用的翻译功能，可以创建一个远程服务实现翻译功能，而不是在每个应用中都实现这个功能。

Android SDK 包括了 Service 类，其中的代码封装了服务的行为。但是 Service 与上面介绍的 AsyncTask 不同，一个 Service 对象不会自动创建自己的线程，而是运行在服务的宿主进程的主线程中。这就意味着如果服务要做一些频繁的 CPU 工作（如 MP3 的回



放或网络操作)就会阻塞主线程,我们应该在这个服务中创建一个新的线程来做这项工作。通过使用一个单独的线程,会减少应用程序不响应(ANR)的错误风险,并且应用程序的主线程能够保留给用户,专用于跟 Activity 的交互。

如果要创建一个服务,有两种方式:启动方式(startService),通过 startService()方法启动;绑定方式(bindService),通过 bindService()方法启动。

应用程序组件可以通过调用 Context.startService()方法获得服务,这个过程也是使服务生效的过程,例如在 Activity 中调用 startService()方法。可以通过调用 Context.startService()启动服务,然后通过调用 Context.stopService()或 Service.stopSelf()停止服务。服务一旦启动,它就能够无限期地在后台运行,即使启动它的组件被销毁。通常,一个被启动的服务只有一个单一操作,并且不给调用者返回结果。例如,这个服务可能在网络上下载或上传文件。当操作完成时,服务应该自己终止。如果仅以启动方式使用的服务,这个服务需要具备自管理的能力,且不需要通过方法调用向外部组件提供数据或功能。

应用程序组件也可以通过调用 bindService()方法启动和绑定一个服务,通过 ServiceConnection 或直接获取服务中的状态和数据信息,例如使用 Activity 的 bindService()方法。被绑定的服务会提供一个允许组件跟服务交互的客户端接口,用于发送请求、获取结果,甚至是跨进程的进程间通信实现远程服务。应用组件绑定服务后,可以使用 ServiceConnection 获取服务对象,并且调用服务中的方法。应用组件通过 Context.bindService()方法绑定服务,并且建立 ServiceConnection;通过 Context.unbindService()方法解除绑定,并且停止 ServiceConnection。如果在绑定过程中服务没有启动,Context.bindService()会自动启动服务。同一个服务可以绑定多个 ServiceConnection,这样可以同时为多个不同的组件提供服务。一个被绑定服务的运行时间与绑定它的应用程序组件一样长。多个组件能够绑定一个服务,但是只有所有这些绑定被解绑后,这个服务才被销毁。

这两种获得服务的方法并不是完全独立的,在某些情况下可以混合使用。例如在 MP3 播放器中,可以通过 Context.startService()方法启动音乐播放的后台服务,但在播放过程中如果用户需要暂停音乐播放,则需要通过 Context.bindService()获取 ServiceConnection 和服务对象,进而通过调用服务对象中的方法暂停音乐播放,并保存相关信息。在这种情况下,如果调用 Context.stopService()并不能够停止 Service,需要在所有的 ServiceConnection 关闭后,Service 才能够真正地停止。

无论使用上述两种方式的一种,还是同时使用这两种方式获得服务,都需要使用到 Intent,这与获得 Activity 组件的方式相同。

6.3.1 服务的生命周期

虽然服务的生命周期比 Activity 的生命周期简单,但服务的生命周期非常重要。因为服务在后台运行,有时用户甚至意识不到它的存在,所以我们更多关注于服务如何创建和销毁。服务的生命周期根据创建一个服务的方式不同而有所不同,见图 6.3,分别是启动方式和绑定方式。

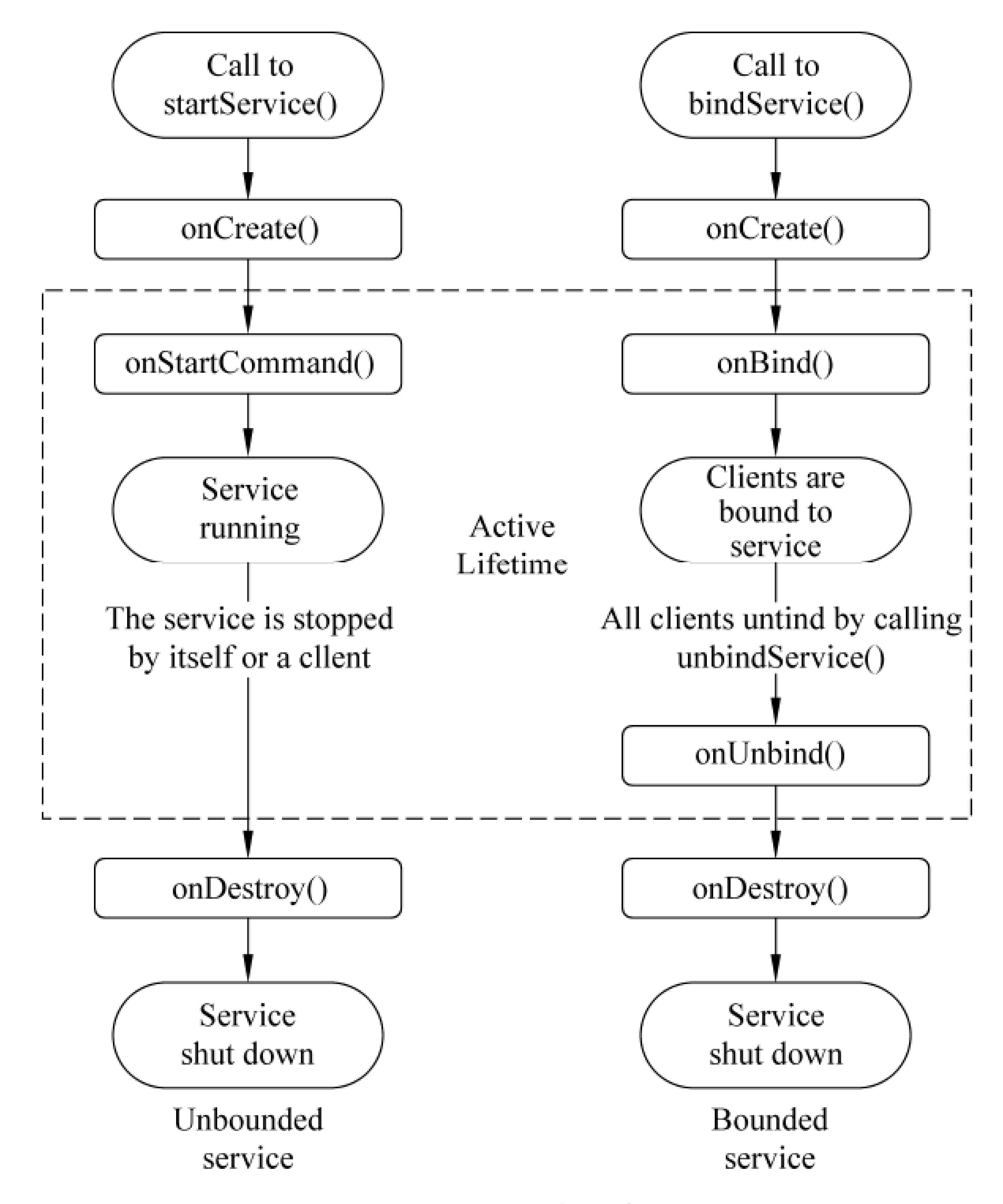


图 6.3　Service 的生命周期

使用启动方式(startService)创建服务时，一个组件调用 startService()方法创建服务，然后服务无限期地运行，并且必须通过调用 stopSelf()方法来终止自己。其他组件也能够通过调用 stopService()方法来终止这个服务。当服务被终止，系统就会把它销毁。

使用绑定方式(bindService)创建服务时，当有一个组件调用 bindService()方法时，服务就会被绑定，客户端通过 IBinder 接口与服务通信。客户端能够调用 unbindService()方法来解除与服务的绑定。可以有多个客户端绑定到一个服务上，但是当所有的绑定都被解除以后，系统才会销毁这个服务，而服务不需要终止自己。

但是，这两种方式是完全独立的，我们能够绑定一个已经用 startService()方法启动的服务。例如，我们可以通过调用 startService()方法启动后台的音乐服务，这个方法使用 Intent 标识了要播放的音乐；之后如果用户想要进行一些播放器的控制时，或想要获取有关当前歌曲信息时，我们可以在一个 Activity 中通过调用 bindService()方法来绑定这个服务。直到所有的客户端解绑后，stopService()或 stopSelf()方法才能实际终止这个服务。

要创建一个服务，必须创建一个 Service 类的子类。Service 实现中，需要重写一些处理服务生命周期关键特征的回调方法，并且给组件提供一种合适的绑定服务的机制。

需要重写的回调方法包括：

- onStartCommand()。当一个组件通过调用 startService()方法请求启动一个服务时，系统会调用这个服务的 onStartCommand()方法。一旦这个方法执行了，那么

这个服务就被启动，并且在后台无限期地运行。实现了这个方法，当服务的工作结束时，必须调用 stopSelf()方法或 stopService()方法来终止服务。如果只让服务提供绑定的能力，不需要实现这个方法。

- onBind()。当一个组件想通过调用 bindService()方法跟这个服务（如执行 RPC）绑定时，系统会调用这个方法。在这个方法的实现中，必须通过返回一个 IBinder 对象给客户提供一个用户跟服务进行交互的接口。这个方法必须实现，但是如果你不允许绑定，那么这个方法应该返回 null。
- onCreate()。当服务被第一次创建时，系统会调用这个方法来执行一次安装过程。这个方法在 onStartCommand()或 onBind()方法之前调用。如果服务正在运行，这个方法就不会被调用。
- onDestroy()。当服务不再使用或正在销毁时，系统会调用这个方法。服务需要使用这个方法来实现一些清理资源的工作，如清理线程、被注册的监听器、接受器等。这是服务能够接受的最后的调用。

如果组件通过调用 startService()方法启动服务，这样会调用服务 onStartCommand()方法，那么这个服务就会一直运行，一直到它自己用 stopSelf()方法终止服务；或另一个组件，通过调用 stopService()方法来终止它。如果一个组件调用 bindService()方法来创建这个服务，并且没有调用 onStartCommand()方法，那么这个服务的运行时间与绑定它的组件运行时间一样长，一旦这个服务从所有的客户端解绑，系统就会销毁它，而不需服务自己或其他组件停止。

Android 系统只有在内存不足，并且为用户提供界面响应而必须释放系统资源时，才会强制终止一个服务。如果服务是被一个正在与用户进行交互的 Activity 绑定，那么它被杀死的可能性很小；如果这个服务被声明运行在前台，那么它也几乎不能被杀死。但是，如果这个服务被启动并且长时间运行，那么随着时间的推移系统会降低它在后台任务列表中位置，并且这个服务将很容易被杀死；如果服务是组件，通过 startService()方法启动了，那么必须把它设计成能够通过系统妥善地处理重启。如果系统为了释放资源杀死了服务，就需要设计当资源可用时怎样有效重启服务，当然这依赖于 onStartCommand()方法的返回值。

像 Activity 一样，服务也有生命周期回调方法，可以通过实现这些回调方法来监测服务内状态的改变，在合适的时机执行工作。代码 6.5 示例了每个生命周期的回调方法。

代码 6.5　ExampleService. java

```
public class ExampleService extends Service {
int mStartMode; //indicates how to behave if the service is killed
IBinder mBinder; //interface for clients that bind
boolean mAllowRebind; //indicates whether onRebind should be used

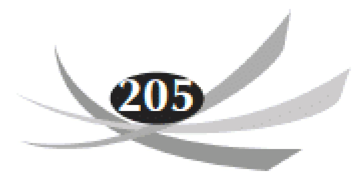
@Override
public void onCreate() {
```

```
//The service is being created
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
//The service is starting, due to a call to startService()
return mStartMode;
}
@Override
public IBinder onBind(Intent intent) {
//A client is binding to the service with bindService()
return mBinder;
}
@Override
public boolean onUnbind(Intent intent) {
//All clients have unbound with unbindService()
return mAllowRebind;
}
@Override
public void onRebind(Intent intent) {
//A client is binding to the service with bindService(),
//after onUnbind() has already been called
}
@Override
public void onDestroy() {
//The service is no longer used and is being destroyed
}
}
```

像 Activity 一样，所有的服务都必须在应用程序的清单文件中声明。要声明服务就要给<application>元素添加一个<service>子元素。例如：

```
<manifest …>
…
<application …>
<service android:name=".ExampleService" />
…
</application>
</manifest>
```

在<service>元素中还包括一些其他的属性定义，如启动服务所需的许可和服务应该运行在哪个进程中。Android:name 属性是唯一必需的属性，它指定了这个服务的类名。一旦应用发布了，就不应该改变这个名字，因为如果修改了，就会中断那些使用 Intent 引用这个服务的功能。关于在清单文件中声明服务的更多信息，请参考<service>



元素的说明。

就像 Activity 一样，一个服务也能够定义 Intent 过滤器，允许其他组件使用隐含的 Intent 来调用这个服务。通过声明 Intent 过滤器，安装在用户设备上的任何应用程序组件都能在符合条件的情况下启动该服务。

6.3.2 创建启动类型服务

Android 的一个组件可以通过调用 startService()方法创建一个启动类型的 Service，并调用服务的 onStartCommand()方法来启动服务。一个服务一旦被启动，它就具有了一个独立于启动它的组件的生命周期，并且这个服务能够无限期地在后台运行，即使启动它的组件被销毁。

Activity 之类的应用程序组件在通过调用 startService()方法来启动 Service 时，需要给指定的 Service 传递一个 Intent 对象，携带一些服务所使用的数据。Service 在 onStartCommand()方法中接受这个 Intent 对象。例如，假设一个 Activity 需要把一些数据保存到在线数据库中。这个 Activity 就能启动一个服务，并且把要保存的数据通过一个 Intent 对象传递给 startService()方法。这个服务在 onStartCommand()方法中接受这个 Intent 对象，连接到互联网，并且执行数据库事务。当事务结束，这个服务就自己终止并销毁。

但是，服务运行的进程与声明它的应用程序的进程是同一个进程，并且是在应用程序的主线程中。默认情况下，如果服务要执行密集或阻塞操作，而用户又要跟同一个应用程序的一个 Activity 进行交互，那么这个服务就会降低 Activity 的性能。要避免影响应用程序的性能，需要在服务的内部启动一个新的线程。

创建启动类型的服务，通常可以通过两种方式来实现：

- 继承 Service。Service 是所有服务的基类。当通过继承这个类创建启动类型服务时，重要的是要给这个服务创建一个新的线程，避免其占用应用程序的主线程，影响正在运行的 Activity 的性能。
- 继承 IntentService。IntentService 是 Service 类的子类，它可以使用工作线程来依次处理所有的启动请求，如果所需创建的服务不用同时处理多个请求，那么这是最好的选择。当通过继承这个类创建启动类型服务时，需要做的所有工作就是实现 onHandleIntent()方法，它接受每个启动请求的 Intent 对象，以便完成后台工作。

1. IntentService

IntentService 是 Service 类的子类，用于处理异步请求。因为大多被启动类型的服务不需要同时处理多个请求，所以使用 IntentService 类来实现自己的服务可能是最好的选择。客户端可以通过 startService(Intent)方法传递请求给 IntentService。

首先分析 IntentService 源代码，见代码 6.8，它是 IntentService 抽象类的具体定义代码。从 IntentService 的定义中可以看出，IntentService 实际上是 Looper、Handler、Service 的集合体，它不仅有服务的功能，还有消息处理和消息循环的功能。

代码 6.6　IntentService.java

```
import android.content.Intent;
import android.os.*;

public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            onHandleIntent((Intent)msg.obj);
            stopSelf(msg.arg1);
        }
    }

    public IntentService(String name) {
        super();
        mName=name;
    }

    public void setIntentRedelivery(boolean enabled) {
        mRedelivery=enabled;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        HandlerThread thread=new HandlerThread("IntentService["+
        mName+"]");
        thread.start();

        mServiceLooper=thread.getLooper();
        mServiceHandler=new ServiceHandler(mServiceLooper);
    }

    @Override
    public void onStart(Intent intent, int startId) {
```

```
        Message msg=mServiceHandler.obtainMessage();
        msg.arg1=startId;
        msg.obj=intent;
        mServiceHandler.sendMessage(msg);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        onStart(intent, startId);
        return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
    }

    @Override
    public void onDestroy() {
        mServiceLooper.quit();
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    protected abstract void onHandleIntent(Intent intent);
}
```

IntentService 在处理事务时，还采用 Handler 方式，创建一个名叫 ServiceHandler 的内部 Handler，并把它直接绑定到 HandlerThread 所对应的子线程。

ServiceHandler 把处理 Intent 所对应的事务的代码都封装到 onHandleIntent()回调方法中，因此直接实现 onHandleIntent()方法，再在里面根据 Intent 的不同进行不同的事务处理就可以了。另外，IntentService 默认实现了 onBind()方法，返回值为 Null。

继承 IntentService 的好处是处理异步请求的时候可以减少写代码的工作量，能比较轻松地实现项目的需求。IntentService 的构造方法一定是参数为空的构造方法，然后再在其中调用父类的构造方法 super("name")。因为 Service 的实例化是系统来完成的，而且系统是用参数为空的构造方法来实例化 Service 的，所以我们只需要实现 onHandleIntent()方法，来完成由客户提供的工作。

下面是一个使用 IntentService 创建启动服务的简单例子，在 onHandleIntent()方法中实现了模拟下载文件时耗时的状况，具体实现代码见代码 6.7。

代码 6.7　HelloIntentService.java

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super IntentService(String)
```

```
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns,
     IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      //Normally we would do some work here, like download a file.
      //For our sample, we just sleep for 5 seconds.
      long endTime=System.currentTimeMillis()+5*1000;
      while (System.currentTimeMillis()<endTime) {
          synchronized (this) {
              try {
                  wait(endTime-System.currentTimeMillis());
              } catch(Exception e) {
              }
          }
      }
  }
}
```

代码 6.7 中只是定义了一个构造方法和 onHandleIntent()方法，如果还要重写 onCreate()、onStartCommand()或 onDestroy()等其他回调方法时，必须要调用父类相同的回调方法，以便 IntentService 对象能够适当地处理工作线程的活动。例如，要在代码 6.7 中添加 onStartCommand()方法的实现代码，则在这个方法的实现代码中，必须执行语句 super. onStartCommand()来调用父类的同一回调方法，使得本类的 onStartCommand()方法获取 Intent 并将其交付给 onHandleIntent()方法。

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
return super.onStartCommand(intent,flags,startId);
}
```

除了 onHandleIntent()方法以外，唯一不需要调用实现的方法是 onBind()方法，但是如果服务允许绑定，就要实现这个方法。

2. Service

继承 IntentService 类来实现一个被启动类型的服务很简单，但是如果服务要执行多线程，而不是通过工作队列来处理启动请求，那么就需要定义 Service 类的子类来处理每个 Intent。

为便于比较，在代码 6.8 所示的例子中，使用继承 Service 类的方式创建了一个服务，执行了与上节继承 IntentService 类的代码 6.7 相同的工作。但与代码 6.7 的处理方式不同，其对于每个启动请求，它都会使用一个工作线程来执行工作，并且每次只处理一个请求。

代码 6.8　HelloService.java

```
public class HelloService extends Service {
  private Looper mServiceLooper;
  private ServiceHandler mServiceHandler;

  //Handler that receives messages from the thread
  private final class ServiceHandler extends Handler {
      public ServiceHandler(Looper looper) {
          super(looper);
      }
      @Override
      public void handleMessage(Message msg) {
          //Normally we would do some work here, like download a file.
          //For our sample, we just sleep for 5 seconds.
          long endTime=System.currentTimeMillis()+5*1000;
          while (System.currentTimeMillis()<endTime) {
              synchronized (this) {
                  try {
                      wait(endTime-System.currentTimeMillis());
                  } catch(Exception e) {
                  }
              }
          }
          //Stop the service using the startId, so that we don't stop
          //the service in the middle of handling another job
          stopSelf(msg.arg1);
      }
  }

  @Override
  public void onCreate() {
    //Start up the thread running the service.Note that we create a
    //separate thread because the service normally runs in the process's
```

```
        //main thread, which we don't want to block.We also make it
        //background priority so CPU-intensive work will not disrupt our UI.
        HandlerThread thread=new HandlerThread("ServiceStartArguments",
                Process.THREAD_PRIORITY_BACKGROUND);
        thread.start();

        //Get the HandlerThread's Looper and use it for our Handler
        mServiceLooper=thread.getLooper();
        mServiceHandler=new ServiceHandler(mServiceLooper);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

        //For each start request, send a message to start a job and deliver the
        //start ID so we know which request we're stopping when we finish the job
        Message msg=mServiceHandler.obtainMessage();
        msg.arg1=startId;
        mServiceHandler.sendMessage(msg);

        //If we get killed, after returning from here, restart
        return START_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        //We don't provide binding, so return null
        return null;
    }

    @Override
    public void onDestroy() {
      Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
    }
}
```

在代码 6.8 中，Service 创建时会调用 onCreate()方法。在 onCreate()方法中创建 Handler 线程(HandlerThread)后启动此线程，然后获取当前线程的 Looper 对象来初始化 Service 的 mServiceLooper，并创建 mServicehandler 对象。

当一个组件通过调用 startService()方法请求启动一个服务时，系统会调用这个服务的 onStartCommand()方法。对于每一个启动 Service 的请求，都会产生一条带有 startId 和 Intent 的消息，并发送到消息队列中。ServiceHandler 通过继承 Handle 来实现服务请



求的多任务处理，具体的原理和细节可以参考6.2节的内容。

继承Service实现服务比继承IntentService类多做很多工作。但是，因为我们自己处理每个onStartCommand()方法的调用，所以就能够同时执行多个请求。在代码6.8所示的例子中没有这么做，但是如果想要这么做的话，那么可以给每个请求创建一个新的线程，并且立即运行它们，而不需要等待前一个请求完成。

3. 启动和终止服务

Android的一个组件可以通过调用startService()方法创建一个启动类型的Service，并调用服务的onStartCommand()方法来启动的服务。

当一个服务定义完成后，可以在其他组件中通过创建符合条件的Intent对象或显示指定要启动的服务，并且将其传递给StartService()方法，这样就可以实现从一个Activity或其他的应用程序组件启动服务。例如，在Activity的onCreate()代码中添加如下代码，能够创建显示指定的Intent对象，并把其作为startService()方法的参数来启动指定的HelloService服务。代码如下：

```
Intent intent=new Intent(this, HelloService.class);
startService(intent);
```

服务一旦运行，就能够使用广播通知(Toast Notification)或状态栏通知(Status Bar Notification)来通知用户。通常，状态栏通知是用来告知后台任务完成得最好的技术(如文件下载完成)，并且用户能够采取相应的动作。

启动类型的服务必须管理它自己的生命周期，除非系统要回收系统内存，否则系统不会终止或销毁这个服务。因此而这种类型的服务必须通过调用stopSelf()方法或另一个组件通过调用stopService()方法才能终止。一旦用stopSelf()方法或stopService()方法请求终止服务，那么系统一有可能就会销毁这个服务。

6.3.3　创建绑定类型服务

绑定类型服务允许组件(如Activity)绑定服务，实现发送请求、接收响应以及执行进程间通信。一个典型的绑定类型的服务只跟它所绑定的应用程序组件同时存在，并且不在后台无限期地运行。绑定类型的服务是客户端和服务端之间交互的服务端。

要创建绑定类型的服务，首先要定义接口，用于指定客户端怎样跟服务进行通信。服务和客户端之间的接口必须是一个IBinder接口的实现，并且要求必须从onBind()回调方法返回这个IBinder接口对象。一旦客户端收到了IBinder对象，就能通过这个接口开始与服务进行交互。多个客户端能够同时绑定这个服务。当客户端完成与服务的交互时，它调用unbindService()方法来解绑。一旦没有客户端绑定这个服务，系统就会销毁它。

客户端通过调用bindService()方法绑定服务。客户端绑定服务时，必须提供ServiceConnection类的对象，并且实现其方法，用来监视服务端的连接。bindService()方法不带有返回值，并立即返回，但是当Android系统在客户端和服务端创建连接时，它会在ServiceConnection上调用onServiceConnected()方法来发送客户端跟服务端进行交互



用的 IBinder 对象。

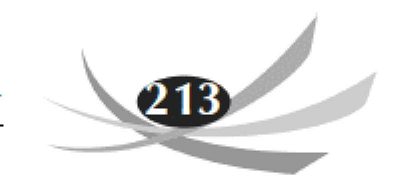
多个客户端能够连接到同一个服务上，但是只在第一个客户端绑定时，系统调用服务的 onBind()方法来发送获取对象。然后系统会给其他任何绑定的客户端发送相同的 IBinder 对象，而不会再次调用 onBind()方法。当最后的客户端从服务上解绑，系统就会销毁这个服务但这个服务也通过 startService()方法启动了的情况除外。

在实现绑定类型的服务时，最重要的部分是定义 onBind()回调方法返回的接口。有很多不同的方法能够定义服务的 IBinder 接口，在这一节下面的内容中会分别讨论这些技术。有三种方法能够定义这个接口。

- 继承 Binder 类。如果我们的服务对应用程序来说是私有服务，并且跟客户端运行在同一个进程中，也就是为本地服务，那么就应该通过继承 Binder 类来创建接口，并且通过 onBind()方法返回这个接口的一个实例。客户端接收这个 Binder 对象，并且能够直接访问其实现的或 Service 中的公共方法。如果只是在后台自己的应用程序提供服务，这是首选方法。只有被其他应用程序或者跨进程使用时，我们才考虑使用其他方法。
- 使用 Messager。如果你的接口要跨越不同进程来进行工作，那么你能用 Messager 给服务创建接口。在这种方式中，服务定义了响应不同消息对象类型的处理器。这个处理器是一个 Messager 的基础，它能够跟客户端共享一个 IBinder 对象，允许客户端使用 Message 对象给服务端发送命令。另外，客户端能够定义一个自己的 Messager，以便服务端能够给客户端发送消息。这是执行进程间通信(IPC)最简单的方法，因为 Messager 队列的所有请求都在一个单线程中，因此不需要针对线程安全来设计你的服务。
- 使用 AIDL。AIDL 为接口定义语言(Android Interface Definition Language)，用来将对象分解成操作系统能够理解的原语，并且将它们在进程之间编组，完成进程间通信。Message 技术实际上是基于 AIDL 架构。就像前面提到的，Message 在一个单线程中创建了一个所有客户端请求的队列，因此服务每次只能接收一个请求。但是，如果想要同时处理多个请求，那么可以直接使用 AIDL。这种情况下，我们的服务必须是多线程的并且要线程安全。要使用直接 AIDL，就必须创建一个定义编程接口的.aidl 文件。Android SDK 使用这个文件生成一个实现文件中定义的接口和处理 IPC 的抽象类，然后能够在服务中进行扩展。大多数应用程序不应该使用 AIDL 方法来创建绑定类型的服务，因为它可能需要多线程的能力，并可能导致更复杂的实现，因此 AIDL 不适用于大多数应用程序。

1. 继承 Binder 类

如果只在应用程序的局部使用服务，并且不需要跨进程工作，程序员可以实现自己的 Binder 类，用它直接为客户端提供访问服务。通常，客户端和服务端只是在同一个应用和进程中工作，不为第三方应用程序提供服务。例如，对于一个需要良好工作的播放音乐的应用程序，就需要把在后台工作的播放音乐的服务与应用的一个 Activity 绑定。使用继承 Binder 类来定义服务的 IBinder 接口的步骤如下：



（1）在 Service 中创建一个 Binder 的实例。

- 这个实例包含 Client 可以调用的 public()方法；
- 这个实例返回当前 Service 对象，其包含 Client 可以调用的 public()方法；
- 这个实例返回 Service 类中的一个类对象，而这个类对象包含 Client 可以调用的 public()方法。

（2）在 Service 的 onBind()方法中返回这个 Binder 实例。

（3）在 Client 端的 onServiceConnected()方法中获得这个 Binder 实例，并通过这个 Binder 实例调用 Service 端的 public()方法。

服务端和客户端必须在同一个应用，原因是客户端能够转换返回的对象，并正确地调用自己的 API。服务端和客户端也必须是在同一个进程中，因为这种技术不执行任何跨进程处理。代码 6.9 是通过继承 Binder 绑定服务的一个简单例子。

代码 6.9　LocalService.java

```
public class LocalService extends Service {
//Binder given to clients
private final IBinder mBinder=new LocalBinder();
//Random number generator
private final Random mGenerator=new Random();

/**
 * Class used for the client Binder. Because we know this service always
 * runs in the same process as its clients, we don't need to deal with IPC.
 */
public class LocalBinder extends Binder {
LocalService getService() {
//Return this instance of LocalService so clients can call public methods
return LocalService.this;
}
}

@Override
public IBinder onBind(Intent intent) {
return mBinder;
}

/** method for clients */
public int getRandomNumber() {
return mGenerator.nextInt(100);
}
}
```

在代码 6.9 中，LocalBinder 对象给客户端提供了 getService()方法，并在 LocalService 的成员变量定义时，通过实例化 IBinder 变量返回当前服务的实例，使得客

户端用这个方法能够获取 LocalService 服务的当前实例。这样就允许客户端调用服务中的 public()方法。

完成服务本身的定义后，第三步就是从客户端绑定定义完成的服务。所谓客户端，就是需要与服务绑定的 Activity 之类的组件。这需要在组件中做几个工作：

- 重写两个回调方法 onServiceConnected() 和 OnServiceDisconnected()，实现 ServiceConnection()。
- 调用 bindService()，传给它 ServiceConnection 的实现。
- 使用接口定义的方法们调用 Service。
- 在需要与 Service 断开绑定连接时，调用 unbindService()。

代码 6.10 是一个例子，Activity 代码绑定了 LocalService 服务，并且在单击一个按钮时调用了服务 public()方法：getRandomNumber()方法。

代码 6.10　BindingActivity.java

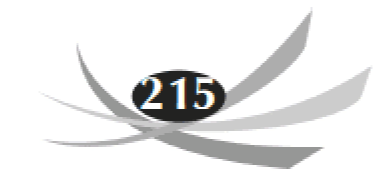
```
public class BindingActivity extends Activity {
LocalService mService;
boolean mBound=false;

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
}

@Override
protected void onStart() {
super.onStart();
//Bind to LocalService
Intent intent=new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
super.onStop();
//Unbind from the service
if(mBound) {
unbindService(mConnection);
mBound=false;
}
}

/** Called when a button is clicked (the button in the layout file attaches to
```




```
 * this method with the android:onClick attribute) */
public void onButtonClick(View v) {
if(mBound) {
//Call a method from the LocalService.
//However, if this call were something that might hang, then this
request should
//occur in a separate thread to avoid slowing down the activity performance.
int num=mService.getRandomNumber();
Toast.makeText(this, "number: "+num, Toast.LENGTH_SHORT).show();
}
}

/** Defines callbacks for service binding, passed to bindService() */
private ServiceConnection mConnection=new ServiceConnection() {

@Override
public void onServiceConnected(ComponentName className,
IBinder service) {
//We've bound to LocalService, cast the IBinder and get LocalService instance
LocalBinder binder= (LocalBinder) service;
mService=binder.getService();
mBound=true;
}

@Override
public void onServiceDisconnected(ComponentName arg0) {
mBound=false;
}
};
}
```

代码6.10显示了客户端怎样使用ServiceConnection接口和onServiceConnected回调方法来绑定服务的。

2. 使用 Messenger

如果服务需要与远程进程通信，就可以使用Messenger对象来给服务提供接口。使用这种方式定义服务与客户端的接口需要的步骤如下：

(1) 在Service内部实现Handler接口，用于处理从每一个Client发送过来的请求。

(2) 使用该Handler创建一个Messenger对象。

(3) 该Messenger在Service的onBind()方法中创建一个IBinder实例，返回给Client。

(4) Client使用从Service返回的IBinder实例来初始化一个Messenger，然后用其给Service发送Message对象。

（5）Service 在它的处理器（Handler）的 handleMessage（）方法中依次接收每个 Message 对象，进行处理。

使用这种方法，客户端没有调用服务端的任何方法，相反客户端会给发送服务端 Message 对象，服务端会在它的处理器中接收这些消息对象。

下面是一个例子，说明了如何使用 Messenger 接口定义绑定的服务，见代码 6.11。

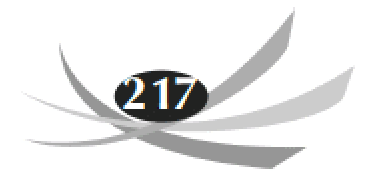
代码 6.11　MessengerService.java

```
public class MessengerService extends Service {
/** Command to the service to display a message * /
static final int MSG_SAY_HELLO=1;

/**
 * Handler of incoming messages from clients.
 * /
class IncomingHandler extends Handler {
@Override
public void handleMessage(Message msg) {
switch(msg.what) {
case MSG_SAY_HELLO:
Toast.makeText (getApplicationContext (), "hello!", Toast. LENGTH_SHORT).
show();
break;
default:
super.handleMessage(msg);
}
}
}

/**
 * Target we publish for clients to send messages to IncomingHandler.
 * /
final Messenger mMessenger=new Messenger(new IncomingHandler());

/**
 * When binding to the service, we return an interface to our messenger
 * for sending messages to the service.
 * /
@Override
public IBinder onBind(Intent intent) {
Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).
show();
return mMessenger.getBinder();
}
}
```



代码6.11中IncomingHandler类中定义的handleMessage()方法，能够接受客户端输入Message对象，并根据Message的what成员属性做出判断。客户端需要做的所有工作就是基于服务端返回的IBinder对象创建一个Messenger对象，并且使用该Messenger对象的send()方法发送消息。

代码6.12是一个简单的Activity代码，它作为客户端绑定服务，并且给服务发送MSG_SAY_HELLO消息。

代码6.12　ActivityMessenger.java

```
public class ActivityMessenger extends Activity {
/** Messenger for communicating with the service. */
Messenger mService=null;

/** Flag indicating whether we have called bind on the service. */
boolean mBound;

/**
 * Class for interacting with the main interface of the service.
 */
private ServiceConnection mConnection=new ServiceConnection() {
public void onServiceConnected(ComponentName className, IBinder service) {
//This is called when the connection with the service has been
//established, giving us the object we can use to
//interact with the service. We are communicating with the
//service using a Messenger, so here we get a client-side
//representation of that from the raw IBinder object.
mService=new Messenger(service);
mBound=true;
}

public void onServiceDisconnected(ComponentName className) {
//This is called when the connection with the service has been
//unexpectedly disconnected--that is, its process crashed.
mService=null;
mBound=false;
}
};

public void sayHello(View v) {
if(!mBound) return;
//Create and send a message to the service, using a supported 'what' value
Message msg=Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
try {
mService.send(msg);
```

```
} catch(RemoteException e) {
e.printStackTrace();
}
}

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
}

@Override
protected void onStart() {
super.onStart();
//Bind to the service
bindService(new Intent(this, MessengerService.class), mConnection,
Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
super.onStop();
//Unbind from the service
if(mBound) {
unbindService(mConnection);
mBound=false;
}
}
}
```

在前面的例子中，通过 Messager 定义接口，实现了 Client 向 Service 发送消息，Service 中的 Handler 对消息做出响应和处理。但这个例子实现的仅是单向通信，即 Client 给 Service 发送消息，并没有实现 Service 处理完成后向 Client 发送消息。如果需要 Service 给 Client 发送消息又该如何实现呢？

这个实现的过程与前面五步类似，只是 Client 和 Service 在实现过程中角色有所变化。可以接着上面的步骤继续来实现双向通信。

(6) 在 Client 中创建一个 Handler 对象，用于处理 Service 发过来的消息。

(7) 使用 Client 中的该 Handler 对象创建一个 Client 自己的 Messenger 对象。

(8) 在前面第(4)步，客户端获取了 Service 的 Messenger 对象，并通过它来给 Service 发送消息。在向 Service 发送消息之前，将 Message 对象的 replyTo 字段设置成第(7)步创建的 Messenger 对象。

(9) Service 的 Handler 处理 Message 时，将 Message 对象的 replyTo 字段提取出



来,并使用其给 Client 发送消息。

这样就实现了 Client 和 Service 的双向通信。Client 和 Service 都有自己的 Handler 和 Messenger 对象,使得对方可以给自己发送消息。Client 的 Messenger 是通过 Message 的 replyTo 传递给 Service 的。

6.4　本章小结

本章主要介绍了 Android 系统多任务的机制和 Service 这个基本的组件。

Android 多任务的调度和实现采用消息驱动机制。应用程序启动时,系统会为它创建一个名为 main 的主线程。默认情况下,同一个应用程序的所有组件都运行在同一个进程的这个"主线程"里。如果希望 Android 应用程序实现多任务,可以通过代码指定组件运行在其他进程里,或为进程创建额外的线程。

Android 线程之间和进程之间是不能直接传递消息的,必须通过对消息队列和消息循环的操作来完成。Android 消息循环是针对线程的,每个线程都可以有自己的消息队列和消息循环。Andriod 提供了 Handler 类和 Looper 类来访问消息队列(Message Queue)。Android 有两种方式实现多线程操作 UI:第一种是创建新线程 Thread,用 Handler 负责线程间的通信和消息;第二种方式是 AsyncTask 异步执行任务。

Service 是 Android 的四大组件之一,用于支持 Android 系统的服务。Service 不能与用户交互,也不能自己启动,需要调用 Context.startService()或 Context.bindService()来启动,在后台运行。Android 的其他应用组件可以在后台启动一个 Service 运行,即使用户切换到另一个应用,此 Service 也会继续运行。无论使用哪一种方式启动,还是同时使用这两种方式获得服务,都需要使用 Intent。

第 7 章 实现数据的存储

任何应用程序都可以通过文件系统存储文件，其他应用程序可以来读取这些文件，当然这可能需要访问权限的设置。Android 提供几种持久化应用程序数据的选择，具体选择哪种方式依赖于具体的需求，例如数据应该是应用程序私有的还是共享的，或者数据所需要的存储空间等。

在 Android 中，应用程序的所有数据对其他应用程序都是私有的，其他应用只有通过设置权限才可以获取数据。Android 系统提供了几种本地数据的存储方式，如果要将这些数据共享，Android 通过定义内容提供器(Content Provider)，能够把私有数据共享给其他应用程序。内容提供器是一种为了开放应用程序的数据读写、具有访问权限的可选组件，可以通过该组件实现私有数据的读写访问。内容提供器提供了请求和修改数据的标准语法和读取返回数据的标准机制。Android 为标准的数据类型提供了一些内容提供器，如图像、视频和音频文件，以及个人通讯录信息。

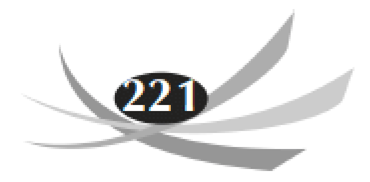
7.1 本地数据存储

Android 的应用程序可以选择的数据存储方式包括以下几种。

- Shared Preferences：用键值对的形式保存私有的原始数据。
- Internal Storage：在设备的内部存储上保存私有的数据，这是内置在设备中不可以任意移除的存储。
- External Storage：在共享的外部存储器上保存公共的数据，这是扩充的存储，可以任意移除。
- SQLite 数据库：在私有的数据库中保存结构化的数据。
- Network Connection：把数据保存在自己的互联网服务器上，例如云存储。本章主要介绍本地数据的存储，所以这部分暂时不介绍。

7.2 Preference 的存取与设置

当应用程序需要保存配置偏好时，不同软件系统都有对应的解决方法。例如 Microsoft Windows 系统通常采用 ini 文件进行保存；J2EE 应用采用 properties 属性文件或者 XML 文件进行保存。Android 提供了一种 Shared Preference 的存储方式，可以称其为共享偏好的存储方式。Shared Preference 存储方式类似 Windows 系统上的 ini 配置



文件，不同之处是它分为多种权限，可以全局共享访问。Shared Preference 方式主要用于保存一些常用的配置，如窗口状态。例如，可以通过它保存上一次用户所做的修改或者自定义参数设定，当再次启动程序后依然保持原有设置。

应用程序通常包括允许用户修改应用程序的特性和行为的设置功能。例如，一些应用程序允许用户指定通知是否启用或指定多久使用云同步数据。如果要为应用程序提供设置 Shared Preferences 的功能（见图 7.1），需要使用 Android 的 Preference APIs 来构建统一的接口。

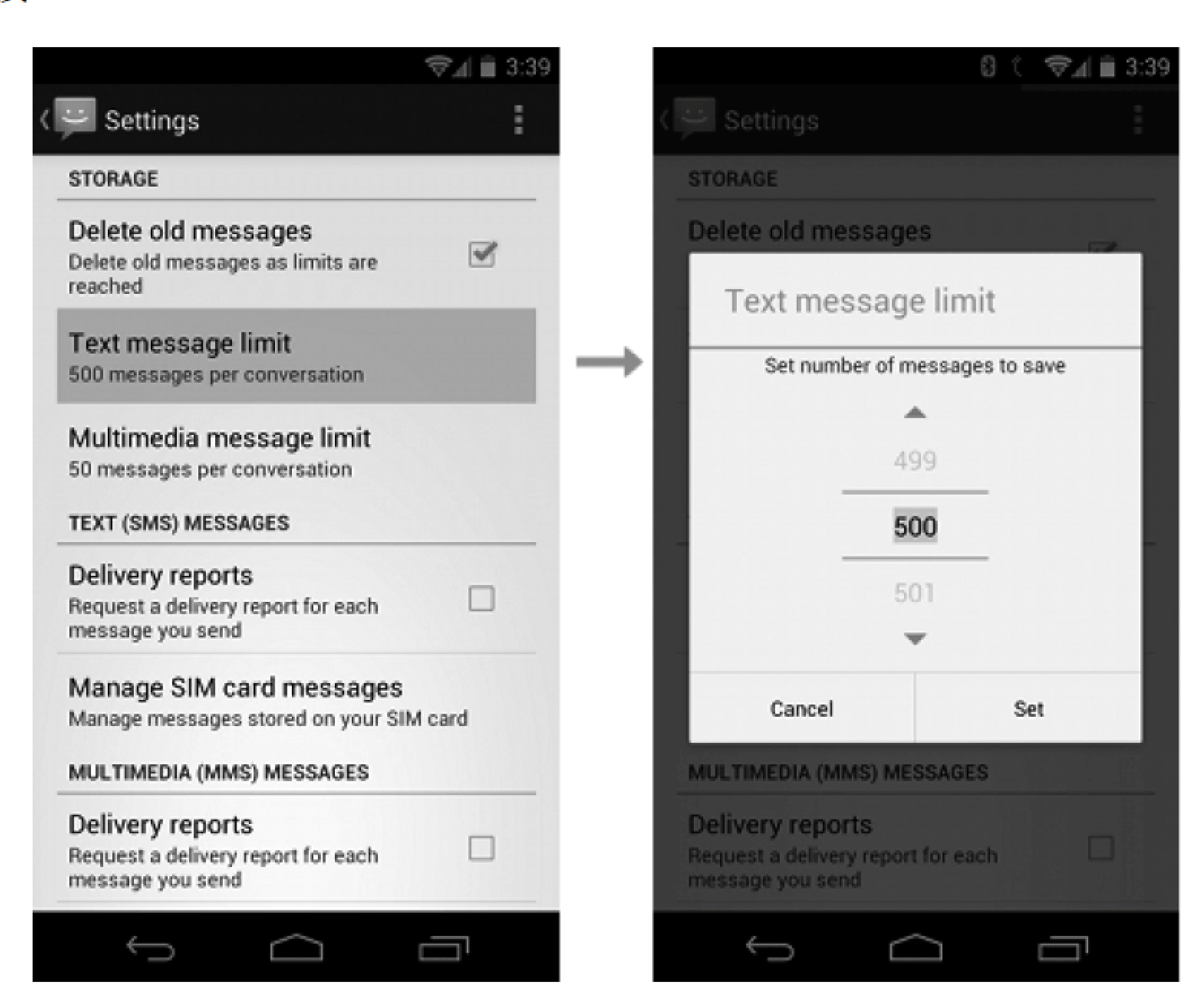


图 7.1　特性和行为的设置

7.2.1　存取 Shared Preferences

Shared Preferences 是一种轻量级机制，其使用键值对能够保存任意类型的原始类型数据：布尔型、浮点型、整数型以及字符串。存储的时候类似于 Map 的 key-Value(键值)对。

Shared Preferences 采用 XML 格式将数据存储到设备中，路径为 data/data/包名/share_prefs/文件名.xml。Shared Preference 处理数据有三种模式：

- MODE_PRIVATE：表示只有创建这个 Preferences 的程序才能访问这个 Preferences。这是默认的模式。
- MODE_WORLD_READABLE：表示其他程序对这个 Shared Preferences 只有只读权限。
- MODE_WORLD_WRITEABLE：表示其他程序同时拥有读写权限。

如果在应用程序中要获取一个 Shared Preferences 对象，有两种方法：

- Context.getSharedPreferences(String name，int mode)：name 为本组件的配置文

件名，mode 为操作模式，默认的模式为 0 或 MODE_PRIVATE。

- Activity.getPreferences(int mode)：这个方法由于没有配置文件名，配置文件仅可以被调用的 Activity 使用。

获取 Shared Preferences 对象后，就可以对其进行读写。

如果要读取 Shared Preferences 中的文件信息，只需要针对文件中所存储的数据类型，直接使用 Shared Preferences 对象的 getXXX()方法，例如 getString()、getInt()或 getBoolean()方法，根据键值对读出数据。

如果要向 Shared Preferences 文件中写入信息，则必须先调用 Shared Preferences 对象的 edit()方法获取一个 SharedPreferences.Editor 对象，使其处于可编辑状态，然后再调用 Editor 对象的 putXXX()方法向文件中按键值对的方式写入数据，最后调用 commit()方法提交更改后的配置文件。

下面是一个简单的例子，设置当前的 Activity 按键静音模式。其采用了 Map 数据结构来存储数据，以键值的方式存储，可以简单地读取与写入，见代码 7.1。

代码 7.1　Calc.java

```
public class Calc extends Activity {
    public static final String PREFS_NAME="MyPrefsFile";

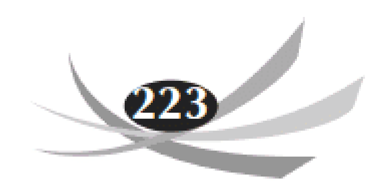
    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       …

       //Restore preferences
       SharedPreferences settings=getSharedPreferences(PREFS_NAME, 0);
       boolean silent=settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
       super.onStop();

      //We need an Editor object to make preference changes.
      //All objects are from android.context.Context
      SharedPreferences settings=getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor=settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      //Commit the edits!
```




```
        editor.commit();
    }
}
```

数据读取与写入的方法都非常简单，只是在写入的时候需要先调用 edit() 获取 SharedPreferences.Editor 对象，然后使用这个对象写入数据，最后使用 commit()提交。Shared Preferences 是采用了 XML 格式将数据存储到设备中，放在 DDMS 中的 File Explorer 中的/data/data/<package name>/shares_prefs 下。

使用 Shared Preferences 是有些限制的：只能在同一个包内使用，不能在不同的包之间使用。

7.2.2　理解 Preference 框架

当开发移动应用时，通常需要存储用户的偏好参数，并且在应用程序运行时使用这些参数。由于这是一个经常性的应用模式，Google 创建了一个偏好框架，提供一种机制使开发者能够很容易地显示、保存和操纵用户的偏好。使用该框架，在 XML 文件中就可以定义丰富的用户界面。可以称这些界面为设置，用来帮助用户选择自己的偏好。

前面章节中讲述的界面控件大多是 View 类的子类，而设置界面是由偏好对象构建的，都是 Preference 的子类。与 View 类界面控件类似，设置界面通过 XML 文件定义不同类型的偏好对象。一个偏好设置界面由一个或多个偏好对象组成。每个偏好对象就是设置界面上的一个项目，为用户提供合适界面，改变偏好设置，例如 CheckBoxPreference 对象是复选框类型的设置界面；而 ListPreference 提供了一个单选的模态窗体。每一个偏好都以键值对的形式保存在应用程序默认的 Shared Preferences 文件中。当用户改变设置时，系统会更新 Shared Preferences 文件中对应的值。读取 Shared Preferences 文件中的数据，可以根据用户的共享参数改变应用程序的行为。

1. 使用 XML 文件定义偏好设置界面

新的偏好对象可以在运行时实例化，也可以在 XML 中用偏好层级对象来定义。使用 XML 定义设置是首选，因为 XML 文件结构更容易阅读，更新也很简单，并且可以在运行时修改它们。每一个偏好子类都能使用 XML 节点来匹配声明。Android 系统预定义了一些偏好选项，下面列出一些主要的选项。

- CheckBoxPreference：是一个带有复选框的偏好项目，可以用来设置打开和关闭两种状态，其在 Shared Preferences 文件中保存的值为 boolean 类型，true 表示打开。
- ListPreference：是一个带有单选按钮的偏好选项，其单选项列表会在一个模态窗体中显示，保存的值支持任何类型。
- EditTextPreference：是一个带有文本编辑框的偏好选项，其保存的值为 String 类型。
- RingtonePreference：用户使用这个偏好选项从设备上选择铃声，选中的铃声 URI 会被保存为 String 类型。

设计 Shared Preferences 设置界面的 XML 文件需要保存在项目的 res/xml 目录下,其文件名可以任意命名,但一般都使用 preferences. xml 作为偏好的文件名。如果要为偏好设置创建多面板布局,则需要为每一个 Fragment 创建单独的 XML 文件。

在偏好设置界面的 XML 文件中,根节点必须是一个<PreferenceScreen>元素,在这个元素中可以定义多个偏好对象。在<PreferenceScreen>中添加的每个子元素代表了偏好设置列表中的一个项目,代码 7.2 是一个 XML 定义偏好设置的示列代码。

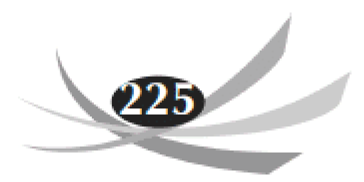
代码 7.2 preferences. xml

```
<?xml version="1.0" encoding="utf-8"?>
< PreferenceScreen xmlns: android =" http://schemas. android. com/apk/res/
android">

    <PreferenceCategory
        android:key="first_category"
        android:title="更新设置">
        <CheckBoxPreference
            android:defaultValue="true"
            android:key="perform_updates"
            android:summary="打开或者关闭数据更新"
            android:title="是否更新" />

        <ListPreference
            android:defaultValue="1000"
            android:dependency="perform_updates"
            android:entries="@array/updateInterval"
            android:entryValues="@array/updateIntervalValues"
            android:key="updates_interval"
            android:summary="定义数据更新的时间间隔"
            android:title="更新间隔" />
    </PreferenceCategory>
    <PreferenceCategory
        android:key="second_category"
        android:title="欢迎信息">
        <EditTextPreference
            android:defaultValue="您好,移动世界"
            android:dialogMessage="请提供欢迎信息"
            android:dialogTitle="欢迎信息"
            android:key="welcome_message"
            android:summary="定义需要显式的欢迎信息"
            android:title="设置欢迎信息" />
    </PreferenceCategory>

</PreferenceScreen>
```



在代码 7.2 中使用了 CheckBoxPreference、ListPreference 和 EditTextPrefence 三种偏好项目，并使用 PreferenceCategory 对偏好项目进行分组。其中包含的一些属性解释如下。

- android:key，当需要保存偏好项目的值时，这个属性是必需的。这个属性为每个偏好对象指定了唯一的 key（一个字符串类型），当系统在 Shared Preferences 文件中保存偏好设置时，需要用到这个属性。
- android:title，为偏好设置提供了一个用户可见的名称。
- android:defaultValue，指定初始值，Android 系统将其保存在 Shared Preferences 文件中。

应用程序中的偏好设置可能定义了一些影响用户操作的重要行为，所以当用户第一次打开应用程序时，有必要为每一个偏好设置项目在 Shared Preferences 文件中赋予初始化默认值。要实现这个功能，首先必须在 XML 文件中为每一个偏好项目指定一个默认值（见代码 7.3）。

代码 7.3　为偏好项目指定默认值

```
<CheckBoxPreference
    android:defaultValue="true"
    … />

<ListPreference
    android:defaultValue="@string/pref_syncConnectionTypes_default"
… />
```

然后，在 Activity 里的 onCreate()方法中调用 setDefaultValues()方法。代码如下：

```
PreferenceManager.setDefaultValues (this, R.xml.advanced _ preferences,
false);
```

在 onCreate()方法中调用 setDefaultValues()方法的目的是为了在应用程序启动的时候使用默认设置初始化。这个方法中有三个参数：第一个参数为应用程序的 Context；第二个为偏好设置 XML 文件的资源 ID；第三个参数为 boolean 值，表示是否多次设置默认值，当然大部分情况下默认值一般只需要设置一次，值为 false 即可。

- android:summary，为偏好设置选项提供一个用户可见的摘要。
- android:entries，只在 ListPreference 中使用，用来定义单选窗口中的显示项目。其引用了一个数组资源@array/updateInterval（见代码 7.4）。

代码 7.4　定义单选窗口的中的显示项目

```
<string-array name="updateInterval">
    <item name="1000">1 秒</item>
    <item name="5000">2 秒</item>
    <item name="30000">30 秒</item>
```

```
    <item name="60000">1 分钟</item>
    <item name="300000">2 分钟</item>
</string-array>
```

- android:entryValues，只在 ListPreference 中使用，用来定义选项的保存值。其引用了另一个数组资源@array/updateIntervalValues（见代码 7.5）。

代码 7.5　定义单选项的保存值

```
<string-array name="updateIntervalValues">
    <item name="1000">1000</item>
    <item name="5000">5000</item>
    <item name="30000">30000</item>
    <item name="60000">60000</item>
    <item name="300000">300000</item>
</string-array>
```

为了在 Acitivity 中显示偏好设置，继承 PreferenceActivity 类，基于偏好对象层级关系来显示一个设置列表。当用户做出一个改变时，PreferenceActivity 能自动保存与每一个 Preference 相关的设置。

与 View 类的 GUI 加载不同，在 onCreate()回调方法中使用 addPreferencesFromResource()调用来添加所定义的 Shared Preferences 设置界面 XML 文件（见代码 7.6）。

代码 7.6　定义 PreferencActivity 子类，引用偏好设置的 XML 文件

```
public class QuickPrefsActivity extends PreferenceActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

如果在 Android 3.0 或更高版本上开发，可使用 PreferenceFragment 来显示 Preference 对象列表，因为 PreferenceFragment 提供比 PreferenceActivity 更为灵活的应用程序结构。使用 PreferenceFragment 需要首先定义一个设置子类 SettingsFragment，然后将这个 Fragment 添加到 Activity 码中（见代码 7.7）

代码 7.7　设置子类 PreferenceFragment，为 Activity 添加定义好的偏好设置

```
static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```




```
        addPreferencesFromResource(R.xml.preferences);
    }
    ...
}

public class SettingsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
    }
}
```

完成偏好设置界面的设计，运行应用程序可以到偏好设置的启动界面（见图 7.2 左图）。然后，选择“是否更新”这个偏好选项设置打开“更新间隔”设置界面（见图 7.2 中图），可以选择具体的更新间隔设置值。这两个设置界面之间的依赖关系是一个非常有用的设定，在 preferences.xml 文件使用 android:dependency="perform_updates"属性来定义了两个项目的依赖关系，这个属性值指向了“是否更新”项目。

选择“设置欢迎信息”，然后会看到对话框，这是一个带有文本编辑框的偏好设置项目（见图 7.2 右图）。对于这种偏好设置选项，其界面上面的显示信息也可用通过 XML 文件定义。在代码 7.2 preferences.xml 中，使用 android:dialogMessage 属性设置对话框的提示信息；使用 android:dialogTitle 属性设置对话框的标题。







图 7.2　偏好设置

默认的情况下，所有应用中的偏好都会保存到一个文件中，调用静态方法 PreferenceManager.getDefaultSharedPreferences()能够获得所保存的偏好值，其返回一个 SharedPreferences 对象，这个对象中包含所有我们在 PreferenceActivity 中使用的偏好对象的键值对。代码 7.8 是从 Shared Preferences 文件中读取偏好项目值的一个简单例子。

代码 7.8　从 Shared Preferences 文件中读取偏好项目值

```
SharedPreferences sharedPref = PreferenceManager. getDefaultSharedPreferences
(this);
String syncConnPref=sharedPref.getString(SettingsActivity.KEY_PREF_SYNC_
CONN, "");
```

通过给偏好项目添加<intent>元素,可以定义需要启动的 Intent,打开一个 Activity 或启动一个服务,例如打开浏览器来查看一个 Web 页面(见代码 7.9)。

代码 7.9　为偏好项目定义 Intent

```
<Preference android:title="@string/prefs_web_page">
    <intent android:action="android.intent.action.VIEW"
            android:data="http://www.example.com" />
</Preference>
```

2. 监听偏好项目的改变

在应用程序运行过程中,有些情况下,当 Shared Preferences 设置发生改变时,希望可以得到通知。这个功能可以使用 SharedPreference. OnSharedPreferenceChangeListener 这个接口来实现。通过调用 registerOnSharedPreferenceChangeListener()方法为 SharedPreferences 对象注册监听器。然后覆盖这个接口的唯一回调方法 onSharedPreferenceChanged()。通过实现这个接口,可以在任意一个偏好项目发生改变时,取得一个回调。代码 7.10 是一个简单的例子。

代码 7.10　定义偏好项目改变监听器

```
public class SettingsActivity extends PreferenceActivity implements
OnSharedPreferenceChangeListener {
    public static final String KEY_PREF_SYNC_CONN="pref_syncConnectionType";
    ...

    public void onSharedPreferenceChanged(SharedPreferences
    sharedPreferences, String key) {
        if(key.equals(KEY_PREF_SYNC_CONN)) {
            Preference connectionPref=findPreference(key);
            //为选中的值设置用户描述摘要。
            connectionPref.setSummary(sharedPreferences.getString
            (key, ""));
        }
    }
}
```

代码 7.10 中,onSharedPreferenceChanged()方法会检测改变的设置是否为一个已知的 preference key。如果是,就会调用 findPreference()来获得改变后的 Preference 对

象，针对其给出所需要的处理。这里设置了一个摘要，用于当用户选中时给出提示信息。其实这是一个比较好的方法，特别是多个被选中时，可以通过现有的 API 让用户知道他们做了些什么并得到反馈。

要让监听器起作用，还需要在 Activity 声明周期中的 onPause()和 onResume()方法中注册和注销的监听器，见代码 7.11。

代码 7.11　注册和注销监听器

```
@Override
protected void onResume() {
    super.onResume();
    getPreferenceScreen().getSharedPreferences()
            .registerOnSharedPreferenceChangeListener(this);
}

@Override
protected void onPause() {
    super.onPause();
    getPreferenceScreen().getSharedPreferences()
            .unregisterOnSharedPreferenceChangeListener(this);
}
```

7.3　文件读取与保存

在应用程序保存文件有两个目的。一种目的只是针对某种应用使用，不能被其他应用使用，其为私有文件，例如电子阅读应用使用的文件具有特殊的格式，只能被电子阅读应用使用。另外一种目的不是针对特定应用的，可以被其他应用使用，其为公共文件，例如照相机产生的图片文件，可以被图像编辑应用或者其他应用使用。当需要对文件操作时，要根据这两个目的正确选择内部存储或是外部存储。一般来说应用程序都会使用两种文件夹，一种是在内部存储上，用来保存私有数据；还有一种是在扩展存储上，用来保存公共数据。

当然，应用程序的私有数据也可以保存在扩展存储上。扩展存储一般是可移动的存储介质，而且缺少安全保护，所以可以用来保存公共数据。移动设备的内部存储由生产厂商固化在设备上，其存储空间是受到限制的，一般不会很大，而且不能扩展。所以，需要考虑将一些文件保存在扩展存储上。具体哪些文件保存在扩展存储上，与文件的大小和用途等多个方面的因素有关。

7.3.1　内部存储

前面学习的 Shared Preferences，主要存储的数据类型是键值对，用于存储简单的信息。而对于按顺序读写的大数据，例如读写镜像文件或基于网络的数据交换，Android 使

用的文件系统与其他平台的基于磁盘的文件系统类似。本节和下一节主要介绍如何使用 File APIs 在 Android 的存储区域中来执行读写 Android 文件系统的操作。

所有的 Android 设备都有两个文件存储区域:内部(Internal)和外部(External)存储器。这两个名称来自早期的 Android,当时大多数设备都提供内置的固定的内存(内置存储器),外加一个可移动的存储介质,如 micro SD 卡(外部存储器)。有些设备把固定不变的存储空间分成内部和外部两部分,这样即使没有可移动的存储介质,也总会有两个存储空间,并且不管外部存储器是可移动的,还是固定的,API 的使用方法是相同的。

Android 的内部存储器与外部存储器各自具有不同的特点,可以根据应用程序的数据存储要求,选择将文件存储到合适的存储区域。表 7.1 是它们的特点对比。

表 7.1 Android 的内部存储器和外部存储器特点对比

内部存储器	外部存储器
始终可访问	并非始终可访问,因为用户可能安装 USB 之类的存储介质,有时会卸载
默认情况下存储的文件只能自己的应用可以访问	是无限制可读的,因此存储的文件可能会被外界应用读取,不可控制
要确保文件不被其他用户或应用访问,内部存储是最好的	当用户卸载应用程序后,系统也会删除此应用程序,使用 getExternalFilesDir()保存的文件
	对于希望与其他应用共享、不需要访问限制的文件来说,外部存储是最好的

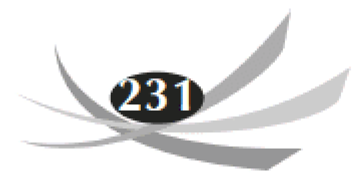
内部存储器(Internal Storage),就是将文件保存在设备内部存储器中。默认情况下,这些文件是相应程序私有的,对其他程序不透明,对用户也是不透明的。当程序卸载后,这些文件就会被删除。在内部存储器上保存数据时,不需要设置任何权限,应用程序始终有权读写它在内部存储器目录中保存的文件。

Android SDK 提供的文件存储权限如下:

- Context.MODE_APPEND,追加方式存储;
- Context.MODE_PRIVATE,私有方式存储,其他应用无法访问;
- Context.MODE_WORLD_READABLE,允许其他应用读取数据;
- Context.MODE_WORLD_WRITEABLE,允许其他应用写入、读取数据。

应用程序的内部存储目录是在 Android 文件系统的特定位置,是由应用程序的包名称来指定的。从技术上来说,如果你把该文件模式设置为可读的,那么另外一个应用程序是可以读取你的内部文件的。但是,其他的应用程序还需要知道你的应用程序的包名和文件名。然而其他的应用程序不能浏览你的应用程序内部目录,而且需要把该文件设置为可读或可写,这样,其他应用程序对该文件的读写条件相当严格。因此在内部存储器上,只要你的文件使用了 MODE_PRIVATE 模式,那么其他的应用程序就不会访问到它们。

一般来说,作为应用程序私有的文件需要保存在内部存储上。例如这些文件可能是下载的杂志、数据文件等,将它们看作是应用程序的一部分。默认情况下,保存在内部存



储上的文件是应用程序的私有数据，其他应用程序不能够访问它们。这些文件存放在应用程序的安装文件夹中，并且遵守一定的命名规则。当用户卸载应用程序时，这些文件也会被删除，这样符合 Android 数据的清理机制。

在内部存储器中创建并保存数据文件，可以按照以下步骤来做：

（1）使用 FileOutputStream 对象，调用 openFileOutput()方法，参数分别为文件名、操作模式，返回值是一个 FileOutputStream。

（2）使用 write()方法向文件中写入数据。

（3）调用 close()方法，关闭输出流。

相关代码见代码 7.12。

代码 7.12　使用内部存储保存文件

```
String FILENAME="hello_file";
String string="hello world!";

FileOutputStream fos=openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

openFileOutput()方法中的参数 FILENAME 用于指定文件名称，不能包含路径分隔符"/"，如果文件不存在，Android 会自动创建它。创建的文件保存的目录为：

```
/data/data/<package name>/files/
```

例如，/data/data/com.google.sample/files/hello_file。"/data/data/com.google.sample/"是应用程序安装时自动产生的目录，用来保存应用程序的私有数据。其中 files 是一个子目录用来保存文件，还有可能有 databases(SQLite 数据库)、shared_prefs(共享引用)、cache(缓存的文件和数据)和 lib(本地库)。

openFileOutput()方法的第二个参数代表文件读写模式。在代码 7.12 中，MODE_PRIVATE 表示要创建一个新文件。如果有同名文件存在，则会替换旧文件，并且这个文件是应用程序的私有文件。其他可用的模式还包括：

- MODE_APPEND：表示只有创建此文件的程序能够使用，其他应用程序不能访问。如果目录中有同名文件，则在原有内容基础上增加数据；
- MODE_WORLD_READABLE：表示可以被其他应用程序读取；
- MODE_WORLD_WRITEABLE：表示可以被其他应用程序写入。

这几个文件读写模式是 Android 定义的常数，值均为整型数字，因此可以使用"|"符号将它们连接起来。下面的代码是一个简单的例子。

```
openFileOutput(FILENAME,
     Context.MODE_APPEND | Context.MODE_WORLD_READABLE);
```

这段代码表示创建的文件可以被其他应用程序读取，而且，如果原来目录中有同名文

件，则在原有文件的基础上增加数据。

如果从内部存储中读取一个文件，需要调用 openFileInput()方法，把要读取的文件名传递给这个方法。具体步骤如下：

（1）调用 openFileInput()方法，参数为即将读取的文件名，该方法返回一个 FileInputStream。

（2）调用 read()方法读取字节。

（3）调用 close()方法关闭输入流。

实现代码见代码 7.13。

代码 7.13　读取内部存储的文件

```
String FILENAME="hello_file";

FileInputStream fis=openFileInput(FILENAME);
byte[] input=new byte[fis.available()];
while(fis.read(input) !=-1){}
String str=new String(input);
fis.close();
```

内部存储中设置的/data/data/<package name>/cache 目录用于放置临时缓存文件。对于一些只需临时缓存的数据，可以使用 Content 类的 getCacheDir()方法来打开一个 File 对象，对这个目录进行操作。如果需要自己创建缓存文件，可以使用 createTempFile()方法创建。例如代码 7.14 中的方法实现了简单的文件读取和临时缓存的功能，首先获取文件名，然后将其保存在内部存储的缓存中。

代码 7.14　文件读取和临时缓存

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName=Uri.parse(url).getLastPathSegment();
        file=File.createTempFile(fileName, null, context.getCacheDir());
    catch(IOException e) {
        //Error while creating file
    }
    return file;
}
```

当设备的内部存储空间不足时，Android 可能会删除这些缓存文件来回收存储空间。但是，最好不要依赖系统来给自动清理这些文件，应该自己来维护缓存文件，把存储空间的耗费限定在合理的范围内，如 1MB。当用户卸载应用程序时，这些文件会被删除。Context 类还提供了其他一些操作目录和文件有用的方法，其中包括：

- getFileDir()：获取保持内部文件的绝对路径，目录的格式为/data/data/<package name>/files；



- getDir()：在内部存储空间中创建或打开一个目录；
- deleteFile()：删除保存在内部存储空间上的文件；
- fileList()：返回当前保存在应用程序中的文件数组列表。

为了避免出现 IOException，通常调用 getFreeSpace()和 getTotalSpace()方法来分别获取存储器的当前剩余的空间和总空间大小。当然也不是必须要调用这些方法，另一种方式是可以在写入文件时捕获 IOException 异常来解决。

在应用程序设计中，注意调用 delete()方法清理不需要的文件。如果文件位于内部存储器上，可以通过 Context 来定位，调用 deleteFile()方法来删除该文件。当用户卸载掉应用程序时，所有保存在内部存储器上的文件，以及通过 getExternalFilesDir()方法保存在外部存储器上的文件都将被删除。而通过 getCacheDir()方法生成的临时文件，需要手动删除。

7.3.2　扩展存储

每个 Android 兼容的设备都支持共享的扩展存储器，用于保存文件。这个存储器可以是一种可移动的存储介质(例如 SD 卡)；或者是不可移动的存储器，被固化在设备里，但是容量要比内部存储大得多。保存在扩展存储上的文件是完全共享的，并且在启用了 USB 存储把文件传输到计算机上时，用户能够修改这些文件。

如果用户将扩展存储挂载到计算机上(见图 7.3)或者移除了这个存储介质，那么手机设备将禁止使用扩展存储。这些保存在扩展存储器上的文件没有安全方面的限制，所有的应用程序都能够读写放在扩展存储器上的文件，而且用户也能够删除它们。



图 7.3　提示 USB 存储设备装载

如果要将文件保存在扩展存储上，首先必须获得扩展存储的权限。要想获得写入外部存储器的权利，需要在 Manifest 文件中取得 WRITE_EXTERNAL_STORAGE 权限。例如：

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_
STORAGE" />
```

目前来说，所有应用程序都有读取扩展存储的权限，而不需特别的权限申明权限。如果需要特别申明对扩展存储的读权限，可以指定 READ_EXTERNAL_STORAGE 权限。当指定了 WRITE_EXTERNAL_STORAGE 权限，默认具有 READ_EXTERNAL_STORAGE 的权限。

对于内部存储，没有必要申明权限，因为应用程序对其在内部存储上私有目录默认就有读写的权限。

在使用扩展存储之前，需要调用 Environment 类的 getExternalStorageState()方法

来检查存储介质是否可用。存储介质可能处于被挂载到计算机上、丢失、只读或者其他状态（见代码 7.15）。

代码 7.15　扩展存储可用性检查

```
boolean mExternalStorageAvailable=false;
boolean mExternalStorageWriteable=false;
String state=Environment.getExternalStorageState();

if(Environment.MEDIA_MOUNTED.equals(state)) {
    //We can read and write the media
    mExternalStorageAvailable=mExternalStorageWriteable=true;
} else if(Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    //We can only read the media
    mExternalStorageAvailable=true;
    mExternalStorageWriteable=false;
} else {
    //Something else is wrong. It may be one of many other states, but all
    we need
    //to know is we can neither read nor write
    mExternalStorageAvailable=mExternalStorageWriteable=false;
}
```

这个例子检查了外部存储器是否可用于读和写。getExternalStorageState()方法还可以用来检查其他状态，例如存储介质是否被共享（连接到一个计算机上）、是否完全丢失、是否被恶意地移除等。在应用程序需要访问存储介质时，要随时获取这些状态，以便给用户提供更多的通知信息。

如果使用 API 级别 8（Froyo Android 2.2）或更高的版本，可以使用 getExternalFilesDir()方法来打开一个 File 对象，获取保存文件的扩展存储目录。这个方法的第一个参数指定了所需的子目录类型。如果此目录类型参数值为 null 值，则返回应用程序在扩展存储中保存文件的如下根目录。代码如下：

```
storage/cdcard0/Android/data/<package_name>/files/
```

<package_name>是 Java 样式的包名，如：com.example.android.app。这是 Android 4.2.2 的目录结构，不同版本目录结构可能不一样。使用 Environment 类中的目录常量可以避免格式不同的问题，能够很好地来传递根目录下子目录的值。例如 DIRECTORY_MUSIC 和 DIRECTORY_RINGTONES 常量分别对应上述文件根目录中的 Music 和 RingTones 子目录，代码如下：

```
File file=new File(
    context.getExternalFilesDir(Environment.DIRECTORY_PICTURES),
    albumName);
```



这个方法会创建与系统规范一致的目录结构。通过指定目录的类型，确保 Android 系统的介质扫描器对文件进行正确的分类。当用户的设备正在运行 API 级别 8 或更高的版本，如果卸载了应用程序，那么这个目录和其所有的内容将会被删除。

如果使用 API 级别 7 或更低的 API 版本，可以使用 getExternalStorageDirectory() 方法来打开一个 File 对象，获取扩展存储的根目录。代码如下：

```
File file=new File(
    Environment.getExternalStorageDirectory(), albumName);
```

但是如果使用这种方法，当应用卸载后这个目录不会清除，会在根目录下留下很多垃圾目录。

如果保存的文件不是应用程序私有的，在应用程序被卸载时需要保留，存储的位置则要放到扩展存储上的公共目录中。这些目录位于外部存储器的根目录，如 Music/、Pictures/、Ringtones/等。

在 API 级别 8 或更高的版本中，可以使用 getExternalStoragePublicDirectory()方法，参数的传递与 getExternalFilesDir()方法类似。对于临时文件，可以使用 getExternalCacheDir()方法来打开一个 File 对象，则在扩展存储目录中保存缓存文件。如果卸载应用程序，这些文件会自动被删除。但是，在应用的生存期间，应该自己管理这些缓存文件，不需要的时候删除这些缓存文件。缓存数据被写入下列目录中：

```
storage/cdcard0/Android/data/<package_name>/cache/
```

7.3.3　文件资源

Android 系统可以定义三种作为资源的文件，它们分别保存在三个不同资源目录中，分别为：/res/xml、/res/raw 和/assets。

发布应用时，如果要把一个静态文件保存到应用程序的发布包中，就要把这个文件保存在项目的 res/raw/目录中。使用 openRawResource()方法可以打开/res/raw/目录中的静态文件，这个方法需要把 R. raw. <filename> 的资源 ID 传递给它。openRawResource()方法会返回一个用于读取文件的 InputStream 对象，但是不能对这个文件进行写入操作。

/res/xml 资源目录可以用来存储 XML 格式的文件，并且和其他资源文件一样，这里的资源是会被编译成二进制格式放到最终的安装包里的，通过 R 类能够访问 XML 文件的资源 ID，并且解析里面的内容。例如，/res/xml 中存放了一个名为 data. xml 的文件(见代码 7.16)。

代码 7.16　data. xml

```
<?xml version="1.0" encoding="utf-8"?>
<book>
    <title>移动电子商务</title>
</book>
```

然后，就可以通过资源 ID 来访问并解析这个文件了（见代码 7.17）。

代码 7.17 使用 PULL 的方法解析 XML 文件

```
XmlResourceParser xml=getResources().getXml(R.xml.data);
    try {
        int eventType=xml.getEventType();
        boolean inTitle=false;
        while (eventType !=XmlPullParser.END_DOCUMENT) {

            //到达 title 节点时标记一下
            if(eventType==XmlPullParser.START_TAG) {
                if(xml.getName().equals("title")) {
                    inTitle=true;
                }
            }
            //如果到达标记的节点则取出内容
            if(eventType==XmlPullParser.TEXT && inTitle) {
                ((TextView) findViewById(R.id.title))
                        .setText(xml.getText());
            }
            eventType=xml.next();
        }
    } catch(XmlPullParserException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    } catch(IOException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    }
```

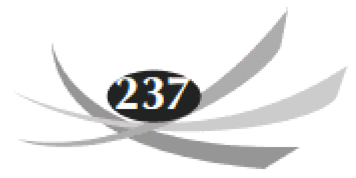
代码 7.17 中，使用 Resources 类的 getXml()方法返回一个 XML 文件的 PULL 解析器。/res/xml 的 XML 文件会被编译成二进制形式。如果想让文件原样存储，必须要存储到/res/raw 资源子目录中，这个目录可以原封不动地将文件存储到设备上，不会被编译为二进制形式，访问的方式也是通过 R 类获得资源 ID。

修改代码 7.17 中的第一句，使用 openRawResource()方法获取资源（见代码 7.18）。

代码 7.18 使用 openRawResource()方法获取 raw 资源

```
XmlPullParser xml=Xml.newPullParser();
        InputStream in=getResources().openRawResource(R.raw.data);
        xml.setInput(new StringReader(in.toString()));
```

代码 7.18 使用 Resource 类中的 openRawResource 方法，返回一个输入流。通过对输入流的操作，就可以任意读取文件中的内容了。



如果需要更高的自由度，尽量不受 Android 系统的约束，那么可以选择使用/assets 资源子目录。这个目录中的文件除了不会被编译成二进制形式之外，另外一点就是访问方式是通过文件名，而不是资源 ID。并且还有更重要的一点就是，可以在这里任意地建立子目录，而/res 目录中的资源文件是不能自行建立子目录的。

使用这种灵活的资源存储方式实现相同的功能(见代码 7.19)。

代码 7.19　访问/assets 文件资源

```
XmlPullParser xml=Xml.newPullParser();
AssetManager asset=getAssets();
InputStream in=asset.open("data.xml");
xml.setInput(new StringReader(in.toString()));
```

代码 7.19 中，使用 Context 类的 getAssets 方法，返回一个 AssetManager 对象；然后使用 open()方法就可以访问需要的资源了，这里 open()方法是打开 assets 目录中的文件，这是 Android 项目资源文件的根目录。所以上面这段代码访问的是 assets 目录中名为 data.xml 文件资源。

7.4　存取结构化数据

Android 中通过 SQLite 数据库引擎来实现结构化数据存储。Android 系统提供了对 SQLite 数据的完全支持，目前支持的版本为 SQLite 3.9。

Android 系统通过 SQLiteDatabase 类来对 SQLite 数据库进行访问，该类封装了一些操作数据库的 API，使用该类可以完成对 SQLite 中数据进行添加(Create)、查询(Retrieve)、更新(Update)和删除(Delete)操作。

7.4.1　SQLite 简介

SQLite 是一款开源的、轻量级的、嵌入式的关系型数据库。它在 2000 年由 D. Richard Hipp 发布，可以支援 Java、.NET、PHP、Ruby、Python、Perl、C 等几乎所有的现代编程语言，支持 Windows、Linux、UNIX、Mac OS、Android、IOS 等几乎所有的主流操作系统平台。目前发布的版本是 SQLite 3.18.0，简称 SQLite 3。

SQLite 是一个嵌入式 SQL 数据库引擎，实现了一个自包含的、无服务器、零配置、事务性的 SQL 数据库，能够针对内存等资源有限的设备(如手机、PDA、MP3)提供一种高效的数据库引擎。目前，SQLite 是部署最广泛的在世界上的 SQL 数据库引擎。

SQLite 具有以下特点：

- 事务处理是原子的、一致的、独立的和持久的(ACID)。
- 零配置，即不需要设置和管理。
- 实现了绝大部分的 SQL92 标准。
- 一个单独的跨平台的磁盘文件存储一个完整的数据库。
- 支持 TB 大小数据库、G 级别的串和二进制大对象。

- 小于 500KB 的代码运行代码空间。
- 对于绝大多数普通操作来说，比流行的 C/S 模式的数据库引擎运行速度快。
- API 简单、易用。
- 自包含，没有外部依赖性。
- 跨平台，支持 UNIX（Linux，Mac OS-X，Android，iOS） and Windows（Win32，WinCE，WinRT）。

SQLite 与大多数其他 SQL 数据库不同，SQLite 没有独立的服务器进程，而是直接对普通的磁盘文件进行读取和写入。一个包含多个表、索引、触发器和视图的完整 SQLite 数据库，全包含在一个单一的磁盘文件中。其数据库文件的格式是跨平台的，可以在 32 位和 64 位系统之间或 big-endian 和 little-endian 体系结构之间进行任意复制。SQLite 的这些特点使其成为应用文件格式的一个普遍选择。

SQLite 由以下几个组件组成：SQL 编译器、内核、后端以及附件（见图 7.4）。SQLite 通过利用虚拟机和虚拟数据库引擎（VDBE），使调试、修改和扩展 SQLite 的内核变得更加方便。

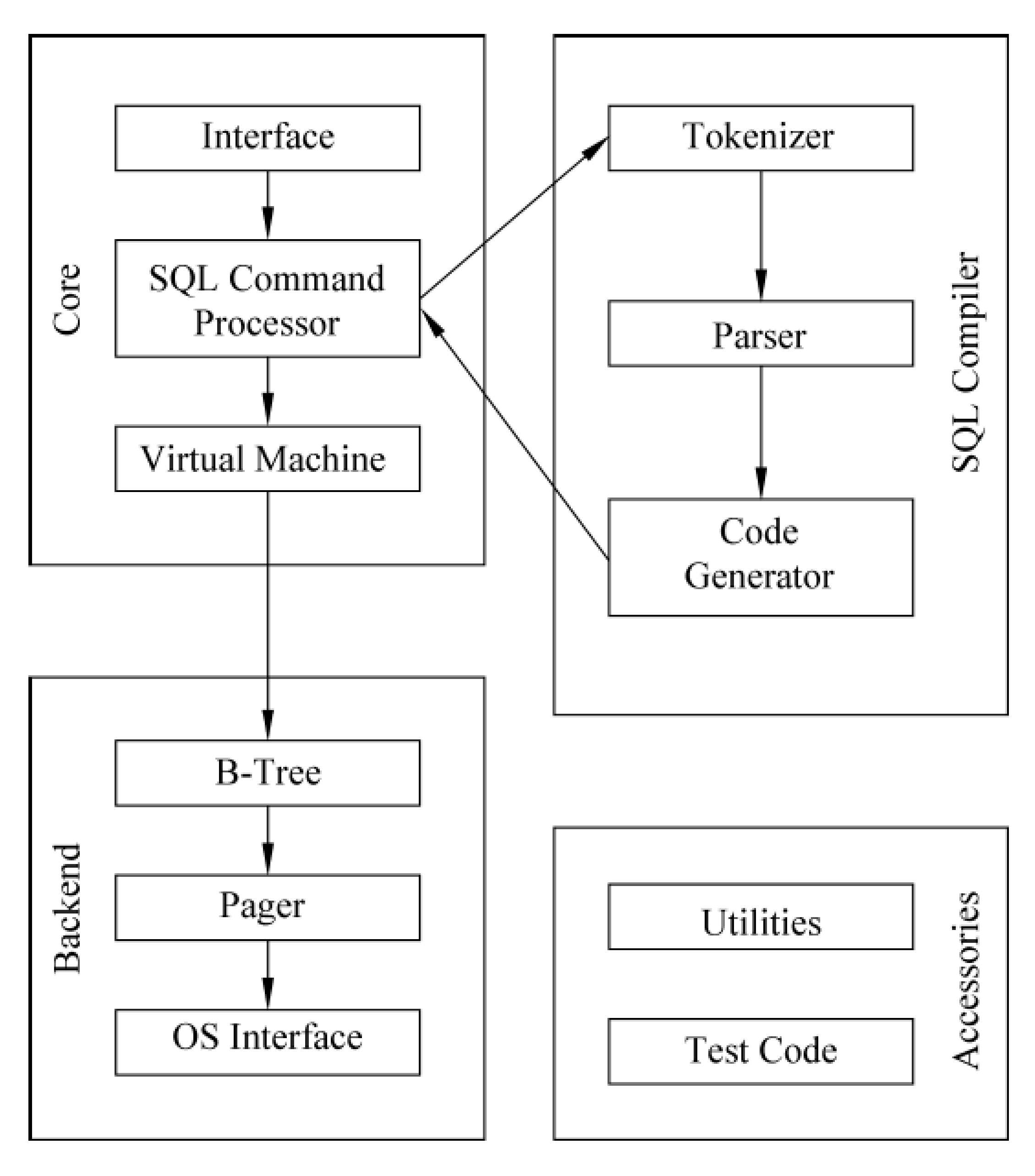
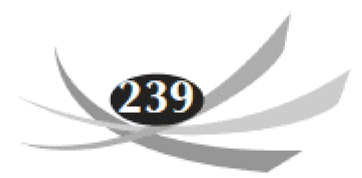


图 7.4　SQLite 的组成

SQLite 3 采用动态类型，支持五种存储的数据类型，包括空值（NULL）、整型（INTEGER）、浮点数（REAL）、字符串（TEXT）和大数据（BLOB）数据类型。虽然它支持的类型虽然只有五种，但也能够接受 VARCHAR(n)、CHAR(n)、DECIMAL(p，s)等数据类型的数据，只是在运算或保存时会转成对应的五种数据类型。

在 SQLite 3 中并没有设定 BOOLEAN 和 DATE 类型，BOOLEAN 型的数据使用 INTEGER 的 0 和 1 代替 true 和 false 来表达，DATE 类型使用特定格式的 TEXT、



REAL 和 INTEGER 的值来代替显示。为了方便 DATE 类型的操作，SQLite 提供了一组函数，详细的使用说明可以参见 http://www.sqlite.org/lang_datefunc.html。这样简单的数据类型设计更加符合嵌入式设备的要求。

SQLite 3 的动态类型特点：不会强制数据类型约束，任何类型的数据可以保存到任何字段中，无论这列声明的数据类型是什么。例如，可以在整型字段中存放字符串，或者在布尔型字段中存放浮点数，或者在字符型字段中存放日期型值。但有一种情况例外，定义为 INTEGER PRIMARY KEY 的字段只能存储 64 位整数，当向这种字段中保存除整数以外的数据时，将会产生错误。

SQLite 遵从标准的 SQL 语句，常用的 SQL 语句语法如下：

(1)查询语句。

```
SELECT * FROM 表名 WHERE 条件表达式 GROUP BY 分组字句 HAVING … ORDER BY 排序子句
```

(2) 插入语句。

```
INSERT INTO 表名(字段列表) VALUES (值列表)
```

(3) 更新语句。

```
UPDATE 表名 SET 字段=表达式[,字段=表达式…] WHERE 条件表达式
```

(4) 删除语句。

```
DELETE FROM 表名 WHERE 条件表达式
```

值得注意的是，SQLite 不支持一些标准的 SQL 功能，特别是外键约束(FOREIGN KEY Constrains)、嵌套 Transaction、RIGHT OUTER JOIN 和 FULL OUTER JOIN，还有一些 ALTER TABLE 功能。

SQLite 是一个紧缩库。如果具有所有的功能，依赖目标平台和编译优化的设置，库的大小可以不超过 500KB，64 位的可能会大一些。如果可选的一些功能省略，SQLite 大小可以低于 300KB。SQLite 可以运行于最小栈空间 4KB 和堆空间 100KB 的环境中，这一优势，使 SQLite 在内存受限的嵌入式领域得到广泛应用。Android 也直接采用了 SQLite 数据库作为结构化存储结构。

Android 系统为 SQLite 数据库的操作提供了 android.database.sqlite 包，用于进行 SQLite 数据库的添加、删除、修改和查询操作，对应用中所创建结构化数据操作提供 API，包括 SQLiteDataBase 和 SQLiteOpenHelper 等类。Android 系统还提供了关系数据库的管理功能，作为支持 SQLite 数据库系统的一部分，通过这些功能可以存取其中的复杂数据集。SQLite 的数据库文件存储的目录为“/data/data/package_name/databases”。

7.4.2 创建 SQLite 数据库

默认情况下，Android 系统下的 SQLite 不具有创建和管理数据库的管理接口或应

用，因此需要通过编码来创建数据库。首先需要创建一个 SQLiteOpenHelper 类的子类来处理数据库所有的操作，例如创建数据库、创建表、插入和删除记录等。

SQLiteOpenHelper 提供有两个方法来操作数据库：

- onCreate(SQLiteDatabase db)：创建数据库时执行，可以在其中执行创建表、字段、视图和触发器等。
- onUpgrade(SQLiteDatabse db, int oldVersion, int newVersion)：当数据库需要更新时执行，需要实现更新版本时表的修改、删除、创建新等操作。

onUpgrade()方法在数据库的版本发生变化时会被调用，数据库的版本是由程序员控制的，假设数据库现在的版本是 Version1.0，由于业务的需要，修改了数据库表的结构，这时候就需要升级软件，升级软件时希望更新用户手机里的数据库表结构，为了实现这一目的，可以把原来的数据库版本设置为 Version 2。并且在 onUpgrade()方法里面实现表结构的更新。当软件的版本升级次数比较多，这时在 onUpgrade()方法里面可以根据原版号和目标版本号进行判断，然后做出相应的表结构及数据更新。onUpgrade()方法在每次成功打开数据库后首先被执行，默认情况下此方法的实现为空。

在创建 SQLiteOpenHelper 类的子类时，需要通过“super (context, name, cursorFactory, version);”调用 SQLiteOpenHelper 的构造方法。其中参数 context 是需要附加到数据库的数据；name 是数据库的名称；cursorFactory 是 Cursor 的一个子类对象，用于查询，可以是空；version 指数据库的版本。代码 7.20 是定义对 SQLite 数据库操作的 SQLiteOpenHelper 子类的模式。

代码 7.20　SQLiteOpenHelper 的子类定义模式

```
public class DatabaseHelper extends SQLiteOpenHelper {
    DatabaseHelper(Context context, String name, CursorFactory
    cursorFactory, int version) {
        super(context, name, cursorFactory, version);
        //TODO SQLiteOpenHelper 构造方法
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            //TODO 创建数据库,对数据库的操作
        }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
    newVersion) {
        //TODO 更新数据库的操作
    }

    @Override
    public void onOpen(SQLiteDatabase db) {
```




```
        super.onOpen(db);
        //TODO 每次成功打开数据库后首先被执行
    }
}
```

下面在 Android 系统中创建一个简单的数据库应用，用来说明如何创建、操作 SQLite 数据库。存储学生的简单信息。其中包括两个表：Students 和 Departments。

两个表的字段定义和相互之间的关系见图 7.5。表 Stdudents 的主键为 StdId，表 Departments 的主键为 DeptId。

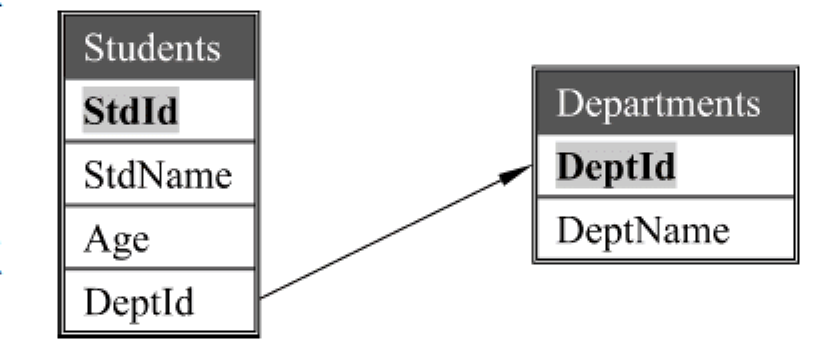


图 7.5　例子中的实体关系图

首先按照代码 7.20 中定义的模式创建一个 SQLiteOpenHelper 子类 DatabaseHelper，并定义一些字符串常量，见代码 7.21。

代码 7.21　定义字符串常量

```
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {

    static final String dbName="demoDB";
    static final String studentTable="Students";
    static final String colID="StdId";
    static final String colName="StdName";
    static final String colAge="Age";
    static final String colDept="DeptId";

    static final String deptTable="Departments";
    static final String colDeptID="DeptId";
    static final String colDeptName="DeptName";

    static final String viewStds="ViewStds";
```

在 Android 系统中创建数据库中的表等对象的代码在 onCreate()方法中执行。代码 7.22 中是 SQLiteOpenHelper 的 onCreate()方法，这里创建了两个表及其字段、一个视图和一个触发器。当数据库创建时，将执行这些代码。也就是当这个数据库在磁盘中不存在时，会在创建它的时候执行一次。这个 onCreate()方法的代码在同一个设备中只执行一次。

代码 7.22　在 onCreate()方法中创建数据库中的对象

```
public void onCreate(SQLiteDatabase db) {
    //TODO Auto-generated method stub

    db.execSQL("CREATE TABLE "+deptTable+" ("+colDeptID
            +" INTEGER PRIMARY KEY , "+colDeptName+" TEXT)");

    db.execSQL("CREATE TABLE "+studentTable+" ("+colID
            +" INTEGER PRIMARY KEY AUTOINCREMENT, "+colName+" TEXT, "
            +colAge+" Integer, "+colDept
            +" INTEGER NOT NULL ,FOREIGN KEY ("+colDept
            +") REFERENCES "+deptTable+" ("+colDeptID+"));");

    db.execSQL("CREATE TRIGGER fk_stddept_deptid "+" BEFORE INSERT "
            +" ON "+studentTable+
            " FOR EACH ROW BEGIN"+" SELECT CASE WHEN ((SELECT "
            +colDeptID+" FROM "+deptTable+" WHERE "+colDeptID
            +"=new."+colDept+") IS NULL)"
            +" THEN RAISE (ABORT,'Foreign Key Violation') END;"+"  END;");

    db.execSQL("CREATE VIEW "+viewStds+" AS SELECT "+studentTable
            +"."+colID+" AS _id,"+" "+studentTable+"."
            +colName+","+" "+studentTable+"."+colAge+","
            +" "+deptTable+"."+colDeptName+""+" FROM "
            +studentTable+" JOIN "+deptTable+" ON "+studentTable
            +"."+colDept+"="+deptTable+"."+colDeptID);
    //Inserts pre-defined departments
    InsertDepts(db);
}
```

Android SDK 包括了 SQLite 数据库工具，可以用这些工具来浏览表的内容，运行 SQL 命令，以及执行其他的 SQLite 数据上的功能。

7.4.3　管理外键约束

数据库完整性(Database Integrity)是指数据库中数据的正确性和相容性。数据库完整性由各种各样的完整性约束来保证的。其中，数据库的外键约束保障了数据库的域完整性和参照完整性。也就是说外键约束实现了表与表之间的联系，外键的取值必须是另一个表的主键，使得在更新、插入或删除操作时，各表间数据保持完整性。

默认情况下，SQLite 3 数据库不支持外键约束。要实现 SQLite 3 中各表的外键约束功能，需要通过定义触发器 Tigger 中的条件来强制实现。

下面使用图 7.5 中 Students 表和 Deparments 表之间外键约束的实现作为例子，说明如何根据数据库设计，在 SQLite 3 中定义外键约束。创建触发器的 SQL 语句如下：

```
CREATE TRIGGER fk_stddept_deptid Before INSERT ON Students
FOR EACH ROW BEGIN
    SELECT CASE WHEN ((SELECT DeptID FROM Dept WHERE DeptID=
                    new.Dept) IS NULL)
    THEN RAISE (ABORT,'Foreign Key Violation') END;
    END
```

在 Android 代码中，使用触发器实现外键约束功能包括两个步骤：

(1) 在 SQLiteOpenHelper 子类的 onCreate()方法中创建触发器。

(2) 覆盖 SQLiteOpenHelper 的 onOpen()方法，激活触发器。

首先，在 onCreate ()方法中创建数据库表之后，定义创建触发器的语句，实现外键约束功能。在代码 7.22 中使用 execSQL()方法实现了上面的创建触发器功能。代码如下：

```
db.execSQL("CREATE TRIGGER fk_stddept_deptid "+" BEFORE INSERT "
        +" ON "+studentTable+
        " FOR EACH ROW BEGIN"+" SELECT CASE WHEN ((SELECT "
        +colDeptID+" FROM "+deptTable+" WHERE "+colDeptID
        +"=new."+colDept+") IS NULL)"
        +" THEN RAISE (ABORT,'Foreign Key Violation') END;"+"  END;");
```

然后，覆盖 onOpen()方法，激活数据库中的外键(见代码 7.23)，强制实现外键约束。

代码 7.23　激活外键，强制实现外键约束

```
@Override
public void onOpen(SQLiteDatabase db) {
    super.onOpen(db);
    if(!db.isReadOnly()) {
      //Enable foreign key constraints
      db.execSQL("PRAGMA foreign_keys=ON;");
    }
}
```

7.4.4　查询和更新 SQLite 数据库

创建 SQLiteDatabase 的表、视图和触发器等对象后，就可以对数据库进行数据的添加、查询、更新和删除操作。使用 SQLiteDatabase 对象执行 SQL 语句的方法 db.execSQL(String statement)，可以执行 INSERT、DELETE、UPDATE 和 CREATE TABLE 之类有更改行为的 SQL 语句；使用 db.rawQuery()和 db.Query()方法，可以执行 SELEDCT 查询语句。除了直接执行 SQL 语句的方法之外，SQLiteDatabase 还专门提供了对应于添加、删除、更新、查询的操作方法：insert()、delete()、update()和 query()。

要查询和更新 SQLite 数据库，首先要打开数据库，建立数据库连接。Android 系统调用 SQLiteOpenHelper 的 getWritableDatabase()或者 getReadableDatabase()方法打开

数据库，获取用于操作数据库的 SQLiteDatabase 实例。如果数据库不存在，Android 系统会自动生成一个数据库，然后调用 onCreate()方法创建数据库表结构及其他数据库对象，并添加应用会使用到的初始化数据，onCreate()方法在初次生成数据库时才会被调用，在 onCreate()方法里可以生成数据库表结构及添加一些应用使用到的初始化数据。

getWritableDatabase()和 getReadableDatabase()方法都可以获取一个用于操作数据库的 SQLiteDatabase 实例。getWritableDatabase()方法以读写方式打开数据库，当数据库的磁盘空间满时，数据库就只能读而不能写，会发生错误，需要进行异常处理。而 getReadableDatabase()方法返回的对象与 getWritableDatabase()类似，但当数据库的磁盘空间满时，以只读方式打开数据库，问题解决后，就可能够以调用 getWritableDatabase()方法重新返回一个可读写的 SQLiteDatabase 数据库对象了。不过 getReadableDatabase()返回需要花费很长时间，最好不要在主线程中调用。

一旦 SQLiteDatabase 实例被缓存，多次调用 getWritableDatabase() 或 getReadableDatabase()方法得到的都是同一实例。如果不再使用数据库，及时使用 close()方法关闭数据库，回收资源。

接下来，在前面创建的数据库基础上，使用简单的例子说明在 Android 系统中，通过 SQLiteDatabase 如何对数据库进行操作。

1. 插入记录。

要使用 SQLiteDatabase 对象进行插入操作，首先要使用 getWritableDatabase() 以可读写的方式打开与数据库的连接，然后使用 ContentValues 类的对象创建一个记录，通过 ContentValues 类的 put()方法给记录逐个字段赋值，最后通过 SQLiteDatabase 的 insert()方法将新记录插入数据库中，关闭数据库(见代码 7.24)。

其中 ContentValues. put(ColumnName，value)参数的有两个，ColumnName 是表的字段名称，value 是新记录的值。

代码 7.24 定义插入记录操作 AddStudent()

```
void AddStudent(Student std) {

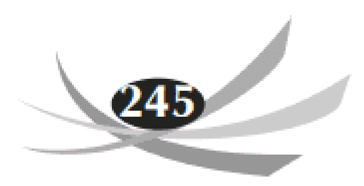
    SQLiteDatabase db=this.getWritableDatabase();

    ContentValues cv=new ContentValues();

    cv.put(colName, std.getName());
    cv.put(colAge, std.getAge());
    cv.put(colDept, std.getDept());
    //cv.put(colDept,2);

    db.insert(studentTable, colName, cv);
    db.close();
}
```

对于 insert()方法本身来说，无论第三个参数是否包含数据，执行 insert()方法一定



会添加一条新记录。但前面设置了外键约束，这里的插入要遵从外键约束的定义。同样后面的更新、删除操作都要遵从外键约束的定义。

2. 更新记录

Android 为执行更新语句提供两种方法：db.execSQL()和 db.update()。

与插入记录类似，要使用 getWritableDatabase() 以可读写的方式打开与数据库的连接，使用 ContentValues 对象创建新记录和赋值，在更新数据库记录时，可以使用 execSQL()方法，直接把 SQL 语句按格式写入参数中，也可以使用 update()方法更新。代码 7.25 中说明了使用 update()方法如何更新记录。

update()方法的参数包括四个：

- String Table：所需更新记录的表名；
- ContentValues cv：赋予新值后的纪录；
- String whereClause：可选 WHERE 子句，格式为 columnName＋"＝?"或 null，null 值则更新所有行；
- String[] whereArgs：WHERE 子句中的参数，把 whereClause 中的"?"替换为具体的值，没有 WHERE 子句则为 null。

代码 7.25　定义更新记录操作 UpdateStd()

```
public int UpdateStd(Student std) {
    SQLiteDatabase db=this.getWritableDatabase();

    ContentValues cv=new ContentValues();
    cv.put(colName, std.getName());
    cv.put(colAge, std.getAge());
    cv.put(colDept, std.getDept());

    return db.update(studentTable, cv, colID+"=?",
        new String[] { String.valueOf(std.getID()) });
}
```

3. 删除记录

Android 为执行删除记录提供两种方法：db.execSQL()和 db.delete()。

与插入和更新 SQLite 数据库类似，使用 getWritableDatabase() 以可读写的方式打开与数据库的连接后，删除记录可以使用 execSQL()方法，也可以使用 delete()方法。代码 7.26 说明了如何使用 delete()方法删除数据库中符合要求的记录。

delete()的参数定义与 insert()类似。

代码 7.26　定义删除记录操作 DeleteStd()

```
public void DeleteStd(Student emp) {
    SQLiteDatabase db=this.getWritableDatabase();

    db.delete(studentTable, colID+"=?",
```

```
        new String[] { String.valueOf(emp.getID()) });
    db.close();
}
```

4. 执行查询

Android 为执行查询记录提供两种方法：db.rawQuery()和 db.query()。

查询记录时，使用 getReadableDatabase() 以可读方式打开与数据库的连接，然后把查询条件以参数形式带入 rawQuery()或 query()方法中执行，返回查询结果集。无论是 rawQuery()还是 query()方法，每个 SQLite 查询都会返回一个 Cursor 对象，它指向查询结果的所有行，即结果集。Cursor 对象始终能够在数据库的查询结果中导航，并且能够读取当前行和列的数据。

代码 7.27 中实现了查询表 Departments 中所有记录的功能，可以看出如何使用 rawQuery()对表进行查询。

代码 7.27 rawQuery()查询

```
Cursor getAllDepts()
{
    SQLiteDatabase db=this.getReadableDatabase();
    Cursor cur=db.rawQuery("SELECT "+colDeptID+" as _id,
                    "+colDeptName+" from "+deptTable,new String [] {});

    return cur;
}
```

rawQuery()的参数包括两个：

- String query：字符串形式的选择语句，其中 WHERE 子句中的条件值使用“?”代替。
- String[] whereArgs：WHERE 子句中“?”对应的具体值，没有 WHERE 子句则为 null。

代码 7.27 中查询没有 WHERE 子句约束。下面的语句是加上 WHERE 子句约束，使用 rawQuery()实现的查询，查询结果是从表 Students 中查询 DeptId 值为 2 并且年龄大于 19 的学生学号和名称。这个例句可以更进一步清楚说明 rawQuery()方法中两个参数的使用。

```
Cursor cur=db.rawQuery("SELECT StdId as _id, StdName FROM Students WHERE
                    DeptId=? AND Age>=?", new String[] {"2", "19"});
```

这个查询也以 Cursor 对象作为返回对象。

在 rawQuery()第一参数的 SELECT 语句中，如果数据表的主键字段的名称不是“_id”，那么需要使用“SELECT 字段名 as _id”子句把主键字段的别名命名成“_id”，因为 Cursor 对象总是把名为“_id”的字段作为主键，否则会抛出异常。

另一种执行查询的方法是使用 db. query()方法，代码 7.28 使用 db. query()实现查询某个系的所有学生的学号、姓名、年龄和系名称。

代码 7.28　db. query()查询

```
public Cursor getStdByDept(String Dept) {
    SQLiteDatabase db=this.getReadableDatabase();
    String[] columns=new String[] { "_id", colName, colAge, colDeptName };
    Cursor c=db.query(viewStds, columns, colDeptName+"=?",
        new String[] { Dept }, null, null, null);
    return c;
}
```

db. query()方法中的参数，按顺序分别为：

- String TableName：表名；
- String [ ] columns：查询结果的字段；
- String [ ] whereArgs：WHERE 子句参数；
- String GroupBy：分组子句；
- String Having：HAVING 子句；
- String OrderBy：排序子句；

7.4.5　管理游标 Cursor

游标 Cursor 是数据库系统为用户开设的一个数据缓冲区，存放 SQL 语句查询的执行结果。每个游标区都有一个名字，游标能从包括多条数据记录的结果集中每次提取一条记录进行处理。用户不仅可以用 SQL 语句逐一从游标中获取记录，并赋给主变量，交由主语言进一步处理。游标还允许应用程序对查询结果集中每一行进行相同或不同的操作，而不是一次对整个结果集进行同一种操作。游标提供对基于游标位置而对表中数据进行删除或更新的功能。

在 Android 系统中，每个 SQLite 查询都会返回一个 Cursor 对象，它指向查询结果的所有行，即结果集。Cursor 类提供许多方法，可以根据需要访问结果集中的记录。

Cursor 游标常用方法：

- getCount()：获得总的数据项数；
- isFirst()：判断是否第一条记录；
- isLast()：判断是否最后一条记录；
- moveToFirst()：移动到第一条记录；
- moveToLast()：移动到最后一条记录；
- move(int offset)：移动到指定记录；
- moveToNext()：移动到下一条记录；
- moveToPrevious()：移动到上一条记录；
- getColumnIndexOrThrow(String columnName)：根据列名称获得列索引；
- getInt(int columnIndex)：获得指定列索引的 int 类型值；

- getString(int columnIndex)：获得指定列缩影的 String 类型值。

对于代码 7.28 的查询结果，可以通过 Cursor 对象的访问，输出到控制台（见代码 7.29）。

代码 7.29　结果及输出

```
public void printQuery(Cursor c) {

    if(cursor.moveToFirst() {   //判断游标是否为空
        for(int i=0;i<cursor.getCount();i++){   //遍历游标
            cursor.move(i);
            int stdId=cursor.getInt(0);
            String stdName=cursor.getString(1);
            Int stdAge=cursor.getInt(2);
            String deptName=cursor.getString(3);

            //输出用户信息
            System.out.println(stdId+" : "+stdName+" : "+stdAge+" : "+
            deptName+"\n");
        }
    }
}
```

7.5 本章小结

本章主要介绍了 Android 系统的数据存储机制和存取方法。Android 的应用程序可以选择的本地数据存储方式包括 Shared Preferences、Internal Storage、External Storage 和 SQLite 数据库。

Shared Preferences 保存用户配置偏好的存储方式，使用键值对的形式保存私有的原始数据，以 XML 格式将数据存储到设备中。

Android 的应用程序都会使用两种文件夹，一种是在内部存储(Internal Storage)上，用来保存私有数据；还有一种是在外部存储(External Storage)上，用来保存公共数据。默认情况下，内部存储文件是相应程序私有的，对其他程序不透明，对用户也是不透明的，当程序卸载后，这些文件就会被删除。外部存储的文件可能会被外界应用读取，不可控制。Android 系统可以定义三种作为资源的文件，分别保存在三个不同资源目录中，分别为：/res/xml、/res/raw 和/assets。

Android 中通过 SQLite 数据库引擎来实现结构化数据存储。Android 系统通过 SQLiteDatabase 类来对 SQLite 数据库进行访问，该类封装了一些操作数据库的 API，使用该类可以完成对 SQLite 中数据进行添加(Create)、查询(Retrieve)、更新(Update)和删除(Delete)操作。

第 8 章

内容提供者

任何应用程序都可以通过文件系统或数据库存储文件，其他应用程序可以来读取这些文件（当然可能需要某些访问权限的设置）。在 Android 上，应用程序的所有数据对其他应用程序都是私有的，其他应用只有通过设置权限才可以获取数据。Android 系统提供了几种本地数据的存储方式，如果要将这些数据共享，Android 通过定义 ContentProvider，能够把私有数据公开给其他应用程序。

ContentProvider 是一种为了开放应用程序的数据读写，具有访问权限的可选组件，可以通过这个组件实现私有数据的读写访问，译为中文是“内容提供者”。ContentProvider 提供了请求和修改数据的标准语法和读取返回数据的标准机制。Android 为标准的数据类型提供了一些 ContentProvider，如图像，视频和音频文件，以及个人通讯录信息。Android 提供几种持久化应用程序数据的选择，具体选择哪种方式依赖于具体的需求，例如数据应该是应用程序私有的还是共享的，或者数据所需要的存储空间等。

8.1 ContentProvider 基础

ContentProvider 是 Android 系统提供给用户的一个接口，用于管理如何访问应用程序私有数据的存储库。这里的数据包括结构化存储数据和非结构化存储数据。

8.1.1 什么是 ContentProvider

第 7 章我们提到过，一般情况下，一个 Android 应用程序的数据是私有的，其他应用程序不具有访问的权限。但很多时候 Android 应用程序的服务包括数据的服务，某些数据希望开放给其他 Android 应用程序使用。Android 系统解决这个问题的方法是定义一个通用的、格式统一的数据访问接口，使其他应用程序可以通过调用这个接口提供的方法，访问和修改数据。

Android 系统所定义的访问数据资源的接口称为 ContentProvider。ContentProvider 主要是被其他应用程序引用，为应用程序提供一个一致的、标准的数据访问接口，其中包含了处理进程间的联系和数据安全访问。应用程序向外部开放数据访问，都通过 ContentProvider 来实现。

ContentProvider 向外部应用程序呈现的数据就像一张二维表，就像是在关系数据库里一样。每行显示一些数据类型的实例，每列显示实例数据集合的字段。例如在

Android 平台上有一个内置的 ContentProvider 用户字典，存储了用户想保存的非标准词的拼写，表 8.1 显示了数据在 ContentProvider 的表中可能看起来的样子。

表 8.1　ContentProvider 的表中数据

Word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
Applet	user2	225	fr_CA	3
Const	user1	225	fr_BR	4
Int	user5	100	en_Uk	5

在表 8.1 中，每行代表了一个不能在标准字典中找到的词，每一列代表了这个词的一个属性。第一行是存储在 ContentProvider 中的列名称。在这个 ContentProvider 中，_ID 列作为"主键"列，由 ContentProvider 自动管理维护。

主键对于一个 ContentProvider 并不是必须具备的，即使有主键，ContentProvider 也不必一定要使用_ID 作为主键的列名。但是，如果要把 ContentProvider 中的数据通过用户界面显示出来，常常需要把 ContentProvider 绑定到一个叫作 ListView 的用户界面控件上，这就必须有一个列名叫做_ID。在后面讨论显示查询结果的部分，会对此有详细的解释。

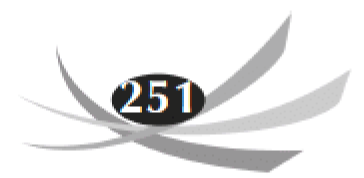
8.1.2　访问提供器 ContentResolver

应用程序使用 ContentResolver 客户端对象来访问 ContentProvider 的数据，可以称其为访问提供器。ContentResolver 的方法提供了基本的 CRUD（创建，检索，更新和删除）数据存储的功能。

应用程序通过 ContentResolver 的对象访问 ContentProvider 时，所使用的方法会调用 ContentProvider 一个具体子类对象的相同名字的方法。例如，为了从用户字典的 ContentProvider 中获得单词和它们出现的语言环境列表，可以使用 ContentResolver.query()方法（见代码 8.1）。这个 query()方法会调用在用户字典 ContentProvider 中定义的 ContentProvider.query()方法。

代码 8.1　ContentResolver.query()的调用

```
//访问用户字典并返回游标
mCursor=getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,       //词表的内容 URI
    mProjection,                            //每行中返回数据的列的名称
                                            //null 表示返回所有列的数据
    mSelectionClause                        //过滤条件
    mSelectionArgs,                         //过滤条件的参数
    mSortOrder);                            //返回行的排序方式
```



ContentProvider.query()方法的参数与 SQL 语言中 SELECT 语句的参数类似，说明如下：

- Uriuri：Content URI，对应 Provider 中的表名；
- String[]projection：查询结果中包含的字段；
- String selection：可选 WHERE 子句，格式为 columnName+“=?”或 null，null 值则更新所有行；
- String[] selectionArgs：WHERE 子句中的参数，把 whereClause 中的“?”替换为具体的值，没有 WHERE 子句则为 null；
- String sortOrder：排序子句。

因此，在使用其他应用程序定义的 Povider 数据时，对于应用程序来说，ContentResolver 的对象只是一个访问器，用于与 Provider 连接以及传递操作和参数，并不需要知道数据的具体存储结构，也不需要编码实现功能，具体的访问操作由 Provider 子类的同名方法来完成。这样，其他应用程序只要了解 Provider 的数据二维表，就可以完成数据交互了。

当然，如果要访问 Provider，应用程序必须在 Manifest 文件中添加特定的权限。这部分内容将在 ContentProvider 权限中详细介绍。

客户端应用程序进程中的 ContentResolver 对象和 ContentProvider 对象会自动处理进程间通信。ContentProvider 也会以二维表的形式，在存储的数据和数据的外部显示之间作为中间的抽象层。

8.1.3　内容统一资源标识

在调用 ContentProvider.query()方法时，第一个参数是 URI 类的对象。URI 是 Android 用来描述 Content URI 的类。

什么是 Content URI 呢？通用资源标志符 Universal Resource Identifier，简称 URI。Content URI 就是 ContentProvider 中数据的内容统一资源标识，能够在存储介质中唯一标识 ContentProvider 中数据所在的具体位置。这有点类似通过 Web 地址去访问网页。ContentProvider 对象通过 URI 来选择要访问 Provider 的表和数据，当调用 ContentResolver 客户端的方法来访问 Provider 中的一个表时，会把这个表对应的 URI 标识作为参数传递给调用的方法。

在 Android 系统中，Content URI 主要分三个部分：scheme、authority 和 path。其中 authority 又分为 host 和 port(见图 8.1)。

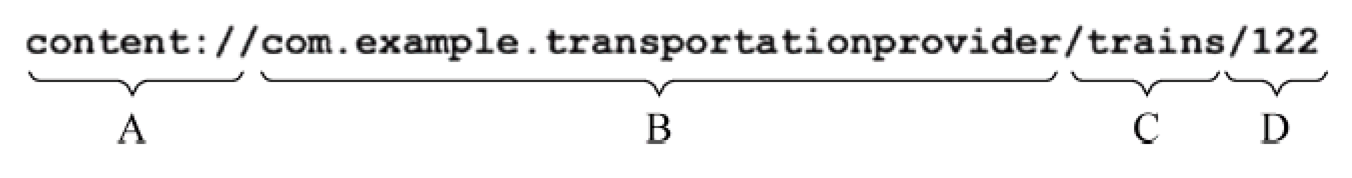


图 8.1　通用资源标识符 URI 结构

一般来说，资源标识符的结构包括三个部分：协议类型、资源名称和路径。下面对照 Internet 的统一资源定位符，分别说明在 Content URI 中这三部分的具体定义。

协议类型：称为 scheme，对应图 8.1 中的 A 部分，表示资源的类型。如果是网站资

源则为“HTTP://”，Content URI 则为“content://”。

资源名称：称为 authority，对应图 8.1 中的 B 部分，表示资源的唯一名称。如果是网站资源则为域名，Content URI 中则是 Provider 的唯一标识名称，可以使用包和类名来定义 Provider 资源名称。

路径：称为 path，对应图 8.1 中的 C 和 D 部分，表示资源中的数据。如果是网站资源则为服务器上网页的路径名，Content URI 中则是 Provider 中的表名或指定表中的记录。例如图 8.1 中 C 和 D 都是用来指定路径，C 一般用来指定数据库中表的名字，这里指出使用 trains 表中的数据，D 可以用来指定数据库中的一条记录，这里指定了 ID 为 122 的记录，如果没有指定 ID，就表示返回全部记录。

8.1.4 MIME 类型

MIME 的英文全称是 Multipurpose Internet Mail Extensions，称为多用途互联网邮件扩展。这是一种互联网标准。在 1992 年，MIME 最早应用于电子邮件系统，后来也应用到浏览器。当用户访问网站资源时，服务器会返回资源的 MIME 类型，浏览器会根据 MIME 的类型调用正确的程序来查看内容。

ContentProvider 也为给定资源定义了 MIME 类型。每个 MIME 类型由两部分组成，前面是数据的大类别（例如 audio 代表声音数据、image 代表图像数据、text 代表文本数据等），后面定义具体的子类别，格式为：大类别/子类别。例如下面是一些常用的 MIME 类型。

- text/html；
- text/css；
- text/xml；
- text/vnd.curl；
- application/pdf；
- application/rtf；
- application/vnd.ms-excel。

Internet 中有一个专门组织 IANA 来确认标准的 MIME 类型，在 IANA 互联网数字分配机构网站上可以看到已注册的类型和子类型的完整列表，网址为 http://www.iana.org/assignments/media-types/。

已注册的主要类型包括 application、audio、example、message、model、multipart、text、video。如果供应商具有专用的数据格式，那么子类型名称将以 vnd 开头，例如微软 Excel 电子表格使用子类型 vnd.ms-excel 标识，而 pdf 被视为一种专用供应商标准，所以对它的标识没有任何供应商特定的前缀。Internet 发展得太快，很多应用程序等不及 IANA 来确认它们使用的 MIME 类型为标准类型。因此它们使用在类别中以 x-开头的方法标识这个类别还没有成为标准，例如 x-gzip，x-tar 等。事实上这些类型运用得很广泛，已经成为了事实标准。只要客户机和服务器共同承认这个 MIME 类型，即使它是不标准的类型也没有关系，客户程序就能根据 MIME 类型，采用具体的处理手段来处理数据。而 Web 服务器和浏览器（包括操作系统）中，默认都设置了标准的和常见的 MIME 类型，只

有对于不常见的 MIME 类型，才需要同时设置服务器和客户浏览器，以进行识别。

Android 遵循类似的约定来定义 MIME 类型，而且每个内容类型的 MIME 类型都具有两种形式：单条记录和多条记录。

对于单条记录，MIME 类型类似于：

```
vnd.android.cursor.item/vnd.yourcompanyname.contenttype;
```

对于多条记录，MIME 类型类似于：

```
vnd.android.cursor.dir/vnd.yourcompanyname.contenttype
```

上面两个 MIME 类型中，vnd. android. cursor. dir 表示返回多行结果；vnd. android. cursor. item 表示返回单行结果，而子类型是指特定的 Provider。Android 内置的 Provider 通常有一个简单的子类型。例如：通讯录的应用中，当创建一个电话号码时，可以设置 MIME 类型为 vnd. android. cursor. item/phone_v2，其中子类型为 phone_v2。

ContentProvider 开发人员可以基于 Provider 资源名称和表名创建它们自己的子类型模式。例如考虑一个包含列车时刻表的 Provider，Provier 的资源名称为 com. example. trains，其包含的表有 Line1、Line2 和 Line3。如果访问表 Line1 的 Content URI 为：

```
content://com.example.trains/Line1
```

则对于表 Line1，Provider 返回 MIME 类型为多条记录：

```
vnd.android.cursor.dir/vnd.example.line1
```

如果访问表 Line2 的 Content URI 为：

```
content://com.example.trains/Line2/5
```

则对于表 Line2 的行 5，Provider 返回的 MIME 类型为单条记录：

```
vnd.android.cursor.item/vnd.example.line2
```

ContentProvider 类提供了两个方法返回 MIME 类型，其中 getType()方法是必须实现的方法；如果使用 ContentProvider 提供文件类型数据，需要实现 getStreamTypes()方法。

8.2　使用 ContentProvider

下面介绍 ContentProvider 的使用方法。

8.2.1　获取数据

Android 应用程序通过 Content URI 定位，来获取 ContentProvider 中所需要的数

据。Content URI 对于 ContentProvider 来说是唯一的，对于开发人员来说也是非常重要的。因此，通常在 ContentProvider 中将 Content URI 定义为常量，方便开发人员的引用。下面是 Android 系统预定义的一些 ContentProvider 的 Content URI 常量。

```
MediaStore.Images.Media.INTERNAL_CONTENT_URI
MediaStore.Images.Media.EXTERNAL_CONTENT_URI
ContactsContract.Contacts.CONTENT_URI
```

这些常量对应的 Content URI 的值如下：

```
content://media/internal/images
content://media/external/images
content://com.android.contacts/contacts/
```

Android 系统的内置通讯录 Provider 使用 ContactsContract.Contacts.CONTENT_URI 常量来标识 Provider 中联系人数据。

有了具体的 Content URI，为了从 Provider 中检索数据，需要两个基本步骤：

步骤一，给提供器申请读访问权限；

步骤二，构造查询代码。

下面以用户字典的 ContentProvider 为例，说明如何从其中获取数据。

(1) 给提供器申请读访问权限。

能够从 ContentProvider 中获取数据的前提，首先是所访问的 ContentProvider 允许其他应用程序的读访问。

从 ContentProvider 中获取数据，应用程序需要“读权限”。这个权限不能在应用程序运行时设置，需要在应用程序的 Manifest 文件中预先声明需要的权限元素。如果指定了这个元素，当用户安装这个应用程序时，系统会隐式地赋予其相应的权限。

Manifest 文件中的权限声明元素是<uses-permission>元素，权限的值从将要访问 ContentProvider 所定义的权限中选择，根据需求指定准确的权限名称。例如，用户字典的 ContentProvider 定义了权限 android.permission.READ_USER_DICTIONARY，作为其可读取的权限。如果应用程序需要从用户字典的 ContentProvider 读取数据，就需要在其 Manifest 文件中声明用户字典的 ContentProvider 的可读取。代码如下：

```
<uses-permission android:name=" android.permission.READ_USER_DICTIONARY"
/>
```

在使用不同的 ContentProvider 时，如果要了解所使用的 ContentProvider 有哪些具体的访问权限和权限确切的名字，可以参考 ContentProvider 的文档。

(2) 构造查询代码。

权限申请完成后，从 ContentProvider 中检索数据的第二步就是构建查询程序。接下来，以用户字典的 ContentProvider 为例，说明如何使用 ContentResolver.query()获取其中的数据。



由于 ContentResolver. query()的参数对应于 SElECT 语句的结构，类似于关系数据库表的查询。因此如果要获取某个 ContentProvider 中的数据，必须要清楚这个 ContentProvider 可以提供什么数据内容，也就是明了这个二维表的结构。这可以从 ContentProvider 的文档中查到。

首先在应用程序中，对应 ContentResolver. query()的参数，声明一些访问用户字典 Provider 所需要的一些变量(见代码 8.2)。

代码 8.2　变量声明

```
public class TryAccessDictionary{

    String[] mProjection={
        UserDictionary.Words._ID, //Contract class constant for the _ID
        column name
        UserDictionary.Words.WORD, //Contract class constant for the word
        column name
        UserDictionary.Words.LOCALE //Contract class constant for the
        locale column name
    };

    //Defines a string to contain the selection clause
    String mSelectionClause=null;

    //Initializes an array to contain selection arguments
    String[] mSelectionArgs={""};
}
```

当调用 ContentResolver. query()方法时，其实际调用了 Provider 中的 ContentProvider. query()方法。ContentResolver. query()方法返回的结果是一个游标对象，下面详细说明其参数的作用。

- Uri uri：对应 Provider 中的表名对应的 Content URI。
- String[]projection：查询结果中包含的字段。这是一个数组，指定需要返回数据的列，如果为 null 则返回所有列。从效率上来说，如果不是用到所有的列，最好明确指定。
- String selection：指定过滤数据的条件，其格式相当于 SQL 选择语句中的 WHERE 子句的条件表达式，如果为 null，则表示返回所有行。一般来说查询数据的 SQL 表达式是有 WHERE 条件的，而 ContentResolver. query()方法中的 selection 参数就对应 WHERE 条件，它是一个逻辑、布尔值、列名、数值的复合表达式。
- String[] selectionArgs：如果在 selection 参数使用了“?”占位符号，表示这个位置需要指定一个条件值。而这个数值是由 selectionArgs 参数指定的。如果 selection 参数中有多个“?”占位符，selectionArgs 参数中字符串数组的顺序与 selection 参数顺序一致。

为什么需要通过这种方式传入条件参数呢？这是为了防止恶意输入 SQL 语句。如果 ContentProvider 管理的数据保存在 SQL 数据库里，假设有外部不可信的数据插入到原始的 SQL 语句中，有可能导致恶意 SQL 输入。

假设条件参数变量定义如下：

```
String mSelectionClause="var="+mUserInput;
```

如果 mUserInput 是一个需要用户输入的变量，这就为插入恶意的 SQL 语句提供了条件，例如用户可以在界面为 mUserInput 变量输入：

```
nothing; DROP TABLE *;
```

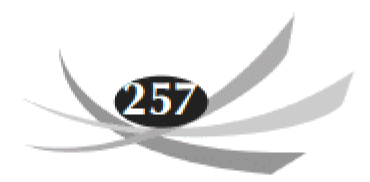
这样在数据库上执行的就不止一条查询语句了，会执行 DROP 操作，就会导致 ContentProvider 删除 SQLite 数据库里所有的表。为了解决这个问题，使用一个带有"?"作为可替代的选择参数，然后再使用另一个选择参数数组来组合定义查询方法。这样，用户的输入会直接绑定到查询方法的选择参数中，而不是作为 SQL 语句的一部分被解释。由于它没有被视为是 SQL 语句，用户输入不可以注入恶意的 SQL。

- String sortOrder：排序子句。指定数据行的排列规则，其格式相当于 SQL 语句的 ORDER BY 子句中的表达式。如果为 null，则使用默认的排序方式，或者不排序。

代码 8.3 定义好变量，就可以编写获取数据的方法了。

代码 8.3　获取 ContentProvider 数据

```
private Cursor getWordDictionary(String mSearchString) {
    String mSelectionClause;
    String[] mSelectionArgs=new String[1];
    if(TextUtils.isEmpty(mSearchString)) {
        mSelectionClause=null;
        mSelectionArgs=null;
    } else {
        mSelectionClause=UserDictionary.Words.WORD+"=?";
        mSelectionArgs[0]=mSearchString;
    }
    Cursor mCursor=getContentResolver().query(
            UserDictionary.Words.CONTENT_URI,
            mProjection,
            mSelectionClause,
            mSelectionArgs,
            mSortOrder);
    return mCursor;
}
```



（3）显示查询结果。

ContentResolver.query()方法执行后，返回的是游标对象，这是一个查询结果集合。如果遍历这个游标对象，就可以读取结果集中的所有数据，输出查询结果。代码8.4中使用Cursor的move()方法在结果集中移动游标指针，使用Cursor的getString()方法获取当前记录各字段的值，并在控制台输出个记录结果。

代码8.4　输出查询结果

```
public void printQueryResult(Cursor c) {

    if(cursor.moveToFirst() {
        for(int i=0;i<cursor.getCount();i++){
            cursor.move(i);
            String word=cursor.getString(0);
            String user=cursor.getString(1);
            String local=cursor.getString(2);

            //输出用户信息
            System.out.println(word+" : "+user+" : "+local+"\n");
    }
}
```

8.2.2　修改数据

如果需要插入、更新或者删除数据，首先要考虑的还是权限的问题。

需要为ContentProvider定义不同的数据访问权限，以便其他应用能够访问其提供的数据。权限的定义可以保证用户能够知道应用程序中的哪些数据可以访问。基于ContentProvider提供的说明，其他的应用程序可以根据自身的需求来申请权限去访问ContentProvider。最后，用户在安装此应用时会看到该应用请求获得的权限。

如果包含ContentProvider的应用没有指定任何权限，其他的应用程序是无法访问该ContentProvider的数据的。但是无论是否指定了权限，包含ContentProvider应用的其他组件拥有对该ContentProvider的完全读写权限。

正如上面所说，在用户字典中使用ContentProvider android.permission.WRITE_USER_DICTIONARY权限来控制对数据的插入、更新和删除。

为获得访问ContentProvider的权限，应用程序在Manifest文件中需要使用uses-permission标签。当Android Package Manager安装应用时，用户必须批准应用程序的所有权限请求。如果用户允许，Package Manager会继续安装流程；如果用户不允许，Package Manager会终止安装。

例如，在应用程序中插入、更新或者删除用户字典ContentProvider的数据，需要在manifest文件中声明android.permission.WRITE_USER_DICTIONARY权限。例如：

```
<uses-permission android:name="android.permission.WRITE_USER_DICTIONARY" />
```

（1）插入数据。

如果向 ContentProvider 中插入数据，需要调用 ContentResolver. insert()方法。这个方法向 ContentProvider 中插入一行新数据，然后返回该行数据的资源标识符。

ContentResolver. insert()方法的参数说明如下：

- Uri url：资源标识符 Content URI；
- ContentValues values：ContentValues 对象，存有所要插入的新记录各字段的值。

ContentValue 对象的使用与第7章相同，使用 ContentValue. put()分别给每个字段赋值，前一个参数是 Provider 表定义的字段名称，后一个参数是给这个字段的赋值。

代码 8.5 中实现了给用户字典 Provide 添加一行新记录的功能。首先创建一个新的 ContentValue 对象，通过 put()把值分别赋给各个字段。这条数据中并没有插入_ID 字段，那是因为它会自动地被增加到数据中。Provider 会给每一行数据赋予一个唯一的_ID，而它往往就被看作数据库表中的主键。

代码 8.5　插入数据

```
Uri mNewUri;
//Defines an object to contain the new values to insert
ContentValues mNewValues=new ContentValues();
/*
 * Sets the values of each column and inserts the word. The
 * arguments to the "put" method are "column name" and "value"
 */
mNewValues.put(UserDictionary.Words.APP_ID, "example.user");
mNewValues.put(UserDictionary.Words.LOCALE, "en_US");
mNewValues.put(UserDictionary.Words.WORD, "insert");
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");

mNewUri=getContentResolver().insert(
    UserDictionary.Words.CONTENT_URI,
    mNewValues //the values to insert
);
mCursor=getWordDictionary(null);
mCursorAdapter.changeCursor(mCursor);
Toast.makeText(this, "插入数据为"+mNewUri.getEncodedPath(),
                    Toast.LENGTH_SHORT).show();
```

ContentResolver. insert()方法的返回值为新增加行的资源标识符，格式如下：

```
content://user_dictionary/words/<id_value>
```

其中的 id_value 为新增行的_ID。Android 系统还提供了自动检测资源标识符格式的 API，例如调用 ContentUris. parseId()方法，返回资源标识符的_ID 值。

(2) 更新数据。

如果要更新 Provider 中的数据，需要使用 ContentResolver. update()方法。与插入数据类似，使用 ContentValues 对象来存储更新数据，同时与查询语句相同的条件参数。如果仅仅需要更新某些字段，只需要把这些字段的值添加到 ContentValues 对象中。如果需要清除一列的值，则把这列设为 null。

ContentResolver. update()方法的参数说明如下：

- Uri uri：Content URI。
- ContentValues values：带有记录更新值的 ContentValues 对象。
- String where：WHERE 子句，具体的条件值使用“?”替代。
- String[] selectionArgs：WHERE 子句中的参数，把 whereClause 中的“?”替换为具体的值，没有 WHERE 子句则为 null。

ContentResolver. update()方法用来定义要更新的列和更新的值，如果我们要清除某一列的内容，使用 ContentValues. putNull()方法将此值设为 null。

代码 8.6 的例子中实现了对用户字典 Provider 表中记录的条件更新，选出“word”字段值中以“en”开头的记录，把其“local”字段更新为 null。

代码 8.6　更新数据

```
//Defines an object to contain the updated values
ContentValues mUpdateValues=new ContentValues();

//Defines selection criteria for the rows you want to update
String mSelectionClause=UserDictionary.Words.LOCALE+"LIKE ?";
String[] mSelectionArgs={"en_%"};

//Defines a variable to contain the number of updated rows
int mRowsUpdated=0;
/*
 * Sets the updated value and updates the selected words.
 */
mUpdateValues.putNull(UserDictionary.Words.LOCALE);

mRowsUpdated=getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,  //the user dictionary content URI
    mUpdateValues                      //the columns to update
    mSelectionClause                   //the column to select on
    mSelectionArgs                     //the value to compare to
);

mCursor=getWordDictionary(null);
mCursorAdapter.changeCursor(mCursor);
System.out.println("更新行数为"+mRowsUpdated);
```

（3）删除数据。

如果要删除 Provider 中的数据，需要使用 ContentResolver. delete()方法。删除 Provider 中的数据与查询获取数据很类似，delete()方法不需要构造新的记录，只需要指定想删除行的选择条件参数，返回值是删除的行数。

ContentResolver. delete()方法的参数说明如下：

- Uri uri：Content URI。
- String where：WHERE 子句，具体的条件值使用“?”替代。
- String[] selectionArgs：WHERE 子句中的参数，把 whereClause 中的“?”替换为具体的值，没有 WHERE 子句则为 null。

下面的代码删除 word 列中以 in 开头的单词。ContentResolver. delete()方法返回删除的行数。

代码 8.7 的例子中实现了对用户字典 Provider 表中记录的删除，删除“word”字段值中以“in”开头的记录。

代码 8.7　删除数据

```
mSelectionClause=UserDictionary.Words.WORD+" LIKE ?";
mSelectionArgs[0]="in%";
//Defines a variable to contain the number of rows deleted
int mRowsDeleted;
//Deletes the words that match the selection criteria
mRowsDeleted=getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI, //the user dictionary 资源标识符
    mSelectionClause, //the column to select on
    mSelectionArgs //the value to compare to
);
mCursor=getWordDictionary(null);
mCursorAdapter.changeCursor(mCursor);
System.out.println("删除行数为"+mRowsUpdated);
```

（4）批模式。

Android 系统还提供了另外一种操作数据的方法，称为批模式。批模式可以一次在一个表中插入多行，或者插入行到多个表中，或者定义一个事务完成一系列跨处理边界的操作。

如果要通过批模式访问 Provider，需要创建包含 ContentProviderOperation 对象的操作数组，然后通过 ContentResolver. applyBatch()方法，将操作数组派发到 Provider 上执行。ContentResolver. applyBatch()方法。操作数组中的 ContentProviderOperation 对象可以对应不同的表。ContentResolver. applyBatch()方法返回值为一个数组。

ContentResolver. applyBatch()方法的参数说明如下：

- String authority：字符串形式的 Content URI 中表的标识，指向需要操作的表。
- ArrayList<ContentProviderOperation> operations：具体的操作。

代码 8.8 中的例子，通过使用批处理模式对 ContactsContract 插入操作，说明了如何使用 ContentResolver.applyBatch()方法。

ContactsContract 是 Android 为通讯录提供的 Provider。从 Android 2.0(API Level 5)开始，Android 平台提供了一个改进的 Contacts API，以适应一个联系人可以有多个账户的需求，例如手机通讯录和 Gmail 通讯录，两个通讯录中的两条记录可以是同一个人。新的 Contacts API 主要是由 ContactsContract 及其相关的类来管理，联系人数据被放到三张表中：Data、RawContacts 和 Contacts。

ContactsContract.Data 表存储了联系人的详细信息，表中的每一行存储一个特定类型的信息，比如 Email、Address 或 Phone。

ContactsContract.RawContacts 用于关联联系人信息与账号，因为有可能手机的联系人信息是从不同的 Gmail 或者其他地方导入的，为互相区别并方便同步，特引入账号概念。

ContactsContract.Contacts 表中的一行表示一个联系人，它是 RawContacts 表中的一行或多行的数据的组合，这些 RawContacts 表中的行表示同一个人的不同的账户信息。Contacts 中的数据由系统组合 RawContacts 表中的数据自动生成。

代码 8.8 实现了从用户界面获取信息，然后把相应的信息插入通讯录的功能。因为插入涉及 Data、RawContacts 和 Contacts 三个表，所以使用批处理来执行。具体的 UI 界面设计与实现参见第 2 章的知识，这里省略。

代码 8.8　使用批模式操作通讯录

```
protected void createContactEntry() {
    //Get values from UI
    String name=mContactNameEditText.getText().toString();
    String phone=mContactPhoneEditText.getText().toString();
    String email=mContactEmailEditText.getText().toString();
    int phoneType=mContactPhoneTypes.get(
           mContactPhoneTypeSpinner.getSelectedItemPosition());
    int emailType=mContactEmailTypes.get(
           mContactEmailTypeSpinner.getSelectedItemPosition());;

ArrayList<ContentProviderOperation>ops=new ArrayList
<ContentProviderOperation>();

//首先向 RawContacts.CONTENT_URI 执行一个插入
//目的是获取系统返回的 rawContactId
    ops.add(ContentProviderOperation.newInsert(RawContacts.CONTENT_URI)
           .withValue(RawContacts.ACCOUNT_TYPE, mSelectedAccount.getType())
           .withValue(RawContacts.ACCOUNT_NAME,
                 mSelectedAccount.getName())
           .build());
    //往 data 表写入姓名数据
```

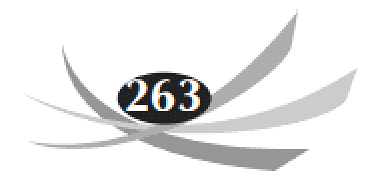
```
    ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
            .withValueBackReference(Data.RAW_CONTACT_ID, 0)
            .withValue(Data.MIMETYPE,
             CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)
            .withValue(CommonDataKinds.StructuredName.DISPLAY_NAME, name)
            .build());
    //往 data 表写入电话数据
    ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
            .withValueBackReference(Data.RAW_CONTACT_ID, 0)
            .withValue(Data.MIMETYPE,
                 CommonDataKinds.Phone.CONTENT_ITEM_TYPE)
            .withValue(CommonDataKinds.Phone.NUMBER, phone)
            .withValue(CommonDataKinds.Phone.TYPE, phoneType)
            .build());
    //往 data 表写入 Email 数据
    ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
            .withValueBackReference(Data.RAW_CONTACT_ID, 0)
            .withValue(Data.MIMETYPE,
                  CommonDataKinds.Email.CONTENT_ITEM_TYPE)
            .withValue(CommonDataKinds.Email.DATA, email)
            .withValue(CommonDataKinds.Email.TYPE, emailType)
            .build());

    try {
       getContentResolver().applyBatch(AUTHORITY, ops);
    } catch(Exception e) {
       //显式警告信息
      Context ctx=getApplicationContext();
      CharSequence txt=getString(R.string.contactCreationFailure);
      int duration=Toast.LENGTH_SHORT;
      Toast toast=Toast.makeText(ctx, txt, duration);
      toast.show();
    }
}
```

在代码 8.8 中，第一部分首先从 UI 界面获取要插入记录的姓名、电话、Email、电话类型和 Email 类型，然后创建新的 ArrayList 对象 ops。ops 变量是一个 ContentProviderOperation 数组。这个数组用来存放多个数据库操作。

如果要想完成一个操作，首先调用 ContentProviderOperation 的 newInsert()方法创建一个构造插入语句的 Builder 对象，然后调用 Builder 中 withValue()方法传入要插入的列和值。

ops 数组一共有四个 ContentProviderOperation 对象。第一个 ContentProviderOperation 对象使用的资源标识符为 ContactsContract.RawContacts.CONTENT_URI，这个资源



表示获得联系人信息的账号(可以保存多个账号的联系人信息,例如 Gmail、本地电话簿等);后面三个 ContentProviderOperation 对象使用的资源标识符为 ContactsContract.Data.CONTENT_URI,这个资源表示联系人的元数据,这里保存了联系人的姓名、电话和 Email。账号和元数据之间具有父子关系,就是如果删除账号,则与其有关联的元数据都要被删除。一般来说,建立这种关联关系的方法是在元数据资源中创建指向账号资源主键的外键。我们通过 withValueBackReference()方法建立这种关系。withValueBackReference()有两个参数:第一个参数是字符串类型,表示子表外键列名;第二参数是 int 类型,表示需要关联 ops 数组中的哪个 ContentProviderOperation 对象,是 ops 数组的索引(从 0 开始的整数)。

ContentProviderOperation 对象中还包括 newAssertQuery()、newDelet()和 newUpdate()等方法,它们分别用来实现查询判定(如果传入期望值,可以判定查询结果与期望值是否相等)、删除和更新的操作。一旦提供了所有的参数后,就可以使用 build()方法创建 ContentProviderOperation 对象。

最后调用 getContentResolver().applyBatch()方法,并且传入资源名称和操作数组对象 ops 执行所有的操作。

8.2.3　预定义的 ContentProvider

Android 系统预先定义了一些 ContentProvider,其中包括:

- Browser:使用 Browser ContentProvider 可以用来读取或修改标签、浏览历史或者网络搜索。
- CallLog:查看或更新电话历史,包括来电和去电、未接来电和电话细节,如联系人和通话时间。
- Contact:使用 ContactProvider 可以用来读取、修改或保持联系人信息。
- MediaStore:提供了对设备上的多媒体文件的集中控制,包括音频、视频和图片。可以在 MediaStore 中保存自己的多媒体来让它可以全局访问。
- Setting:可以使用 SettingProvider 来访问设备的 Preference。使用它,可以查看和修改蓝牙设置、铃声和其他设备设定。

8.3　创建 ContentProvider

前面两节介绍了 ContentProvider 的基础和数据访问,如果要定义一个自己的 ContentProvider,需要实现 ContentProvider 类,并在 Manifest 文件中定义相应的元素。

但是创建 ContentProvider 是一个比较复杂的过程,并非在任何情况下都需要创建 ContentProvider。在构建 ContentProvider 之前,需要考虑一些问题,判断是否有必要创建 Provider。例如需要向其他应用程序提供复杂的数据或文件吗? 需要复制复杂的数据给其他应用程序吗? 需要通过搜索框架提供定制的搜索建议吗? 如果只是在自己的应用中操作 SQLite 数据库,则不需要创建 ContentProvider;如果其他应用程序需要操作这部分数据,则需要创建 ContentProvider。

8.3.1 设计过程

下面是建立一个 ContentProvider 的基本步骤和需要使用的 API，我们还需要定义一个 Activity 来测试 ContentProvider 数据查询和操作。

1. 选择存储结构

ContentProvider 是一个操作结构化数据的接口。在创建接口之前，需要决定如何存储数据。ContentProvider 可以通过两种形式保存数据。一种是使用文件保存，数据通常需要写入文件，如图片、音频、视频。文件存储在应用程序的私有空间里。为了响应其他应用程序的请求，ContentProvider 提供数据文件的句柄。还有一种是使用关系数据库，数据通常存储在数据库、数组或相似的结构中，它们都是以表的行列形式存储数据。行代表一个实体，如一个人或仓库中的一个产品。而列代表这个实体的数据，如人名、产品的价格。在 Android 系统中，这种类型的数据通常是存储在 SQLite 数据库里。在创建 ContentProvider 时，可以根据所存储的数据类型和数据服务，选择适当的数据存储类型。

如果选择使用文件存储数据，Android 系统提供了一系列有关文件操作的 API。如果 ContentProvider 预备提供的数据是位图文件或其他类型面向文件的数据，比较适合把数据存储在一个文件里并且直接提供，而不是通过表提供。其他应用程序在使用这些数据时，需要使用 ContentResolver 文件方法来访问。

如果选择使用关系数据库存储数据，Android 系统提供了包含操作 SQLite 数据库的 API，Android 系统中预定义的 ContentProvider 就是使用关系数据库保存数据。SQLiteOpenHelper 是创建数据库的帮助类，SQLiteDatabase 是访问数据库的基类。虽然 ContentProvider 对外的表现类似关系数据库，但是这对于 ContentProvider 的内部实现来说并不是必需的。为了处理基于网络的数据，还可以使用 java. net 和 android. net 里的 API，把基于网络的数据同步到本地数据存储中，并且以表或文件的形式提供数据。

选择了数据的存储方式之后，一个重要的工作就是设计 ContentProvider 表的数据结构。虽然主键对于一个 ContentProvider 并不是必须具备的，即使有主键，ContentProvider 也不必一定要使用_ID 作为主键的列名。但是，如果要把 ContentProvider 中的数据通过用户界面显示出来，常常需要把 ContentProvider 绑定到一个叫做 ListView 的用户界面控件上，这就必须有一个列名叫做_ID。

在 ContentProvider 支持的数据类型中，Binary Large OBject (BLOB)数据类型用于存储大小变化或数据结构变化的数据。例如，可以使用一个 BLOB 列来存储一个 protocol buffer 或 JSON structure。对于这种类型的数据，可以使用 BLOB 来实现一个独立模式的表，定义一个主键和一个 MIME 类型的列，其他列定义为 BLOB，BLOB 列里的数据意义由 MIME 列来指定。这样我们可以在同一张表里存储不同的数据类型。

2. 定义资源标识符

每一个 ContentProvider 都使用资源标识符 Content URI 来指定其中的数据。通过资源标识符，不仅可以唯一地确定提供数据的 ContentProvider，还可以通过其中的路径来指定 ContentProvider 中的表，甚至可以使用 ID 确切地访问指定表中的唯一行。而且 ContentProvider 中的每个方法都有一个资源标识符作为参数，可以用来确定需要访问的

表、行和文件。因此定义资源标识符是创建 ContentProvider 很重要的部分。

定义资源标识符主要考虑下面几个问题：

(1) ContentProvider 资源名的定义。

在 Android 系统中，ContentProvider 应该具有唯一的资源名，作为其在 Android 里的内部名。为了避免资源名的重复，资源名通常采用域名的格式。由于应用程序的包名也是按照这种格式设计的，因此可以通过扩展包名的方式定义资源名。例如，应用程序的包名为 com. example. <appname>，那么资源名称就可以定义为 com. example. <appname>. provider。

(2) 资源标识符的路径结构。

开发人员通常从资源名开始，在后面追加路径来指向具体的表。例如，在 Provider 中设计了两张表 table1 和 table2，就可以使用下面的路径结构来指定对应的资源：com. example. <appname>. provider/table1 和 com. example. <appname>. provider/table2。路径可以有多个层次，不一定每个层次都指向表。

(3) 处理资源标识符中的 ID。

按照约定，通过使用带有 ID 值的资源标识符可以访问表中指定的一行。这个 ID 在资源标识符的末尾。一般来说，ContentProvider 的 ID 值与表中的_ID 值匹配，可以用来操作对应的数据行。当应用程序访问 ContentProvider 时，这个约定是一个通用的设计模式。应用程序从 ContentProvider 中查询数据返回游标对象，并且利用 CursorAdapter 在 ListView 中显示结果。定义 CursorAdapter 时，需要游标中有一列为_ID。如果用户选取了 ListView 中的一行，希望可以查询或者修改对应的数据，这就需要应用程序从 ListView 的后台游标中得到这行_ID 值，然后附加到资源标识符的后面，然后发送访问请求给 ContentProvider，这样来完成对某一行数据的查询或修改。

3. 资源标识符模式

不同资源标识符的模式对应不同的操作，所以需要识别不同资源标识符的模式。Android 的 API 中包含一个 UriMatcher 类，用来定义不同资源标识符的匹配模式。这个类把资源标识符的模式映射到一个整数，这样应用程序在 switch 语句中可以匹配对应整数来选择对应的操作。在做匹配的过程中，资源标识符模式使用了通配符，其中“*”匹配一个字符串，可以任何长度的任何值；“#”匹配一个字符串，可以是任何长度的数字。

下面举例设计一组资源标识符，并且通过编码处理资源标识符。

假定有一个 ContentProvider 的资源名为 com. example. app. provider，下面的资源标识符则指向具体的表：

```
content://com.example.app.provider/table1: A table called table1
content://com. example. app. provider/table2/dataset1: A table called
dataset1
content://com. example. app. provider/table2/dataset2: A table called
dataset2
content://com.example.app.provider/table3: A table called table3
```

如果在上述的资源标识符后面加上 ID，例如 content://com.example.app.provider/table3/1 则表示表 table3 中主键为 1 的行。对于 com.example.app.provider 来说，以下资源标识符的模式都是可用的：

```
content://com.example.app.provider/*
```

表示匹配 Provider 的任何资源标识符。

```
content://com.example.app.provider/table2/*
```

表示匹配 dataset1 和 dataset2 的资源标识符，不匹配表 table1 或 table3 的资源标识符。

```
content://com.example.app.provider/table3/#
```

表示匹配 table3 中某行的资源标识符。

代码 8.9 示例了 UriMatcher 的方法如何完成模式的匹配。这段代码中，针对表的资源标识符与单行的资源标识符实现了不同的处理方式，其中定义 content://<authority>/<path>模式为表，定义 content://<authority>/<path>/<id>模式为单行。

UriMatcher 的 addURI()方法把资源名和路径映射到一个整数。match()方法返回了对应资源标识符的整数。然后通过一个 switch 语句根据不同整数来对应不同的模式，选择查询表或者单个记录。

addURI()方法的参数说明如下：

- String authority：Content URI 的 authority 部分，也就是资源名。
- String path：Content URI 的 path 部分，详细的表或记录路径。
- int code：模式对应整数。

代码 8.9　定义和使用资源标识符模式

```
public class ExampleProvider extends ContentProvider {
    //定义常量
    private static final int PEOPLE=1;
    private static final int PEOPLE_ID=2;
    private static final int PEOPLE_PHONES=3;
    private static final int PEOPLE_PHONES_ID=4;
    private static final int PEOPLE_CONTACTMETHODS=7;
    private static final int PEOPLE_CONTACTMETHODS_ID=8;

    private static final int DELETED_PEOPLE=20;

    private static final int PHONES=9;
    private static final int PHONES_ID=10;
    private static final int PHONES_FILTER=14;

    private static final int CONTACTMETHODS=18;
```

```
private static final int CONTACTMETHODS_ID=19;

private static final int CALLS=11;
private static final int CALLS_ID=12;
private static final int CALLS_FILTER=15;

private static final UriMatcher sURIMatcher=new UriMatcher(UriMatcher.
NO_MATCH);

//把资源标识符的模式映射到一个整数常量
static
{
    sURIMatcher.addURI("contacts", "people", PEOPLE);
    sURIMatcher.addURI("contacts", "people/#", PEOPLE_ID);
    sURIMatcher.addURI("contacts", "people/#/phones", PEOPLE_PHONES);
    sURIMatcher.addURI("contacts", "people/#/phones/#", PEOPLE_PHONES_
    ID);
    sURIMatcher.addURI("contacts", "people/#/contact_methods", PEOPLE_
    CONTACTMETHODS);
      sURIMatcher.addURI("contacts", "people/#/contact_methods/#",
      PEOPLE_CONTACTMETHODS_ID);
    sURIMatcher.addURI("contacts", "deleted_people", DELETED_PEOPLE);
    sURIMatcher.addURI("contacts", "phones", PHONES);
    sURIMatcher.addURI("contacts", "phones/filter/*", PHONES_FILTER);
    sURIMatcher.addURI("contacts", "phones/#", PHONES_ID);
    sURIMatcher.addURI("contacts", "contact_methods", CONTACTMETHODS);
    sURIMatcher.addURI("contacts", "contact_methods/#", CONTACTMETHODS
    _ID);
    sURIMatcher.addURI("call_log", "calls", CALLS);
    sURIMatcher.addURI("call_log", "calls/filter/*", CALLS_FILTER);
    sURIMatcher.addURI("call_log", "calls/#", CALLS_ID);
}

//使用match(url)获取定义的模式常量,根据常量值返回模式
public String getType(Uri url) {
    int match=sURIMatcher.match(url);
    switch(match)
    {
        case PEOPLE:
            return "vnd.android.cursor.dir/person";
        case PEOPLE_ID:
            return "vnd.android.cursor.item/person";

        …
```

```
                return "vnd.android.cursor.dir/snail-mail";
            case PEOPLE_ADDRESS_ID:
                return "vnd.android.cursor.item/snail-mail";
            default:
                return null;
        }
    }
}
```

另外,ContentUris 类提供了处理资源标识符中 ID 部分的方法,Uri 和 Uri. Builder 类中包含解析 Uri 对象,或者构建新对象的方法。

4. 定义 Contract 类

Contract 类,即合约类,是一个 final public 的类,主要用于定义 Provider 使用的常量,例如资源标识符、表名、列名、MIME 类型和其他一些媒体数据等。Contract 类在 Provider 和其他应用程序之间建立了一个契约,保证 Provider 的数据资源能够被这些程序正确访问。这样,即使 Provider 中的这些常量中的值有变化,也不会影响外部程序的使用。

由于 Contract 类通常使用带有语义的名字来命名常量,可以帮助开发人员减少使用列名或资源标识符的错误,而且还可以包含文档。集成开发环境(例如 Eclipse)可以帮助开发人员选取常量名,并且显示相关文档。

外部程序的开发者从应用程序中不能访问 Contract 类的 class 文件,但在编译时,可以静态编译到应用程序中去。

5. 定义 MIME 类型

ContentProvider 类有两个方法返回 MIME 类型。一个是 getType(),这是必须实现的方法。另一个为 getStreamTypes(),如果 Provider 提供文件类型数据,就需要实现这个方法。

getType()方法返回一个 MIME 格式的字符串,这个字符串描述了资源标识符参数对应的数据类型。资源标识符参数可以是一个具体的标识符,也可以是一个模式。如果参数为模式,则需要返回与这种模式相匹配的资源标识符关联的数据类型。

如果是通常的数据类型,例如 text、HTML 或者 JPEG,getType()方法返回标准的 MIME 类型。这些类型可从官方的网站(http://www.iana.org/assignments/media-types)上查找。

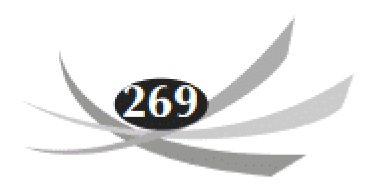
如果是表中一行或多行数据类型,getType()方法返回 Android 特定的 MIME 格式:

(1) type 部分:vnd。

(2) 子类型部分:

- 单行的 URI 模式:android. cursor. item/;
- 多行的 URI 模式:android. cursor. dir/。

(3) Provider 说明的部分:vnd. <name>. <type>。其中 name 值必须是全局唯一



的，type 值必须对应一个资源标识符的模式。name 可以选择公司的名字或应用程序包的部分名字。type 最好标识可以关联资源标识的表。

例如，Provider 的资源名为 com.example.app.provider，表名是 table1，则表示 table1 表里多行数据的 MIME 类型是：

```
vnd.android.cursor.dir/vnd.com.example.provider.table1
```

如果表示 table1 表的单行，MIME 类型是：

```
vnd.android.cursor.item/vnd.com.example.provider.table1
```

如果 Provider 支持的是文件类型数据，需要实现 getStreamTypes()方法。这个方法会根据资源标识符参数从 Provider 中返回包含 MIME 类型的字符串数组。我们可以通过参数来过滤 MIME 类型，仅返回客户端可以处理的 MIME 类型。

例如，假定 Provider 支持.jpg、.png 和.gif 格式的图片文件。当应用程序调用 ContentResolver.getStreamTypes()方法时，如果使用过滤字符串 image/*，表示这是一张图片，ContentProvider.getStreamTypes()方法返回的数组内容为{"image/jpeg","image/png","image/gif"}；如果应用程序仅仅需要文件.jpg，调用 ContentResolver.getStreamTypes()方法时使用过滤字符串*/jpeg，则返回的结果为{"image/jpeg"}。如果 Provider 中没有支持过滤字符串的 MIME 类型，getStreamTypes()方法返回 null。

6. 实现 ContentProvider 的子类

定义自己的 ContentProvider，需要创建 ContentProvider 的子类，使用 ContentProvider 实例来处理其他应用的访问请求，并且管理结构化数据的访问。所有对 Provider 数据的访问，都通过所创建 ContentResolver 对象，调用操作数据的方法，最终调用 ContentProvider 中的具体方法来实现。

因此，在子类里需要代码实现 ContentProvider 提供六个抽象方法，具体来完成对 Provider 的数据操作。这六个抽象方法包括 query()、insert()、delete()、update()、getType()(8.3.1 节已介绍这里不再介绍)和 oncreate()。除了 onCreate()，其他方法都会被访问 ContentProvider 的客户端应用程序调用。

1) query()

用来从 Provider 获取数据。通过参数来选择查询的表、返回行或列、结果排序。方法的查询结果返回游标对象。如果使用 SQLite 数据库存储数据，可以使用 SQLiteDatabase 类的 query()方法来返回游标对象。如果没有匹配的行，也返回游标对象，但是其 getCount()方法返回值为 0。如果在查询中出现内部错误，将返回 null。如果没有使用 SQLite 数据库保存数据，可以使用一个 Cursor 类的具体子类。例如，MatrixCursor 类实现了游标的功能，其中每行数据是数组，可以使用 addRow()方法添加新行。

另外，由于用户程序访问 ContentProvider 是跨进程通信，因此在进程间传递异常信息是非常重要的。IllegalArgumentException 和 NullPointerException 可以在进程间通信，而且对于处理查询异常是非常有帮助的。

2）insert()

用来向 ContentProvider 插入新行。使用参数选择表，获取使用的列值，返回一个新插入行的资源标识符。insert()方法向合适的表里添加行，使用 ContentValues 对象为列设置值。如果 ContentValues 里没有行名，ContentProvider 使用代码里或者数据库框架里的默认值。

这个方法返回新行的资源标识符。使用 withAppendedId()方法将新行的_ID（或其他主键）附加到表的资源标识符后面。

3）delete()

用来删除行。使用参数选择删除的表和行，返回结果为删除的行数。delete()方法没有必要物理地删除行。

4）update()

用来更新存在的行。使用参数选择需要更新的表和行，然后更新其中列的值，返回结果为更新的行数。update()方法使用与 insert()方法相同的 ContentValues 参数，与 delete()和 query()方法使用相同的 selection 和 selectionArgs 参数。这样就可以允许在这些方法之前使用相同的代码。

5）onCreate()

初始化 ContentProvider。Android 系统在创建 ContentProvider 之后就立即调用这个方法。注意直到 ContentResolver 对象需要访问时，ContentProvider 才创建。

由于 Android 系统在 ContentProvider 启动的时候调用 onCreate()方法，因此 onCreate()不能有耗时太多的代码，避免延迟数据库的创建和数据加载。如果在 onCreate()里有耗时太多的任务，会减慢 ContentProvider 的启动，也就会减慢其对其他应用程序的响应。

下面一个例子说明如何实现 ContentProvider 的这些方法。这个例子实现了在方法 ContentProvider. onCreate()里创建一个新的 SQLiteOpenHelper 对象来使用数据库，在打开数据库的时候创建表。这样，第一次调用 getWritableDatabase()时，会自动调用方法 SQLiteOpenHelper. onCreate()。代码 8.10 实现 ContentProvider. onCreate()方法。

代码 8.10　实现 ContentProvider. onCreate()方法

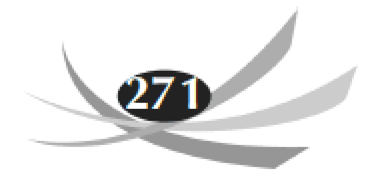
```
public class ExampleProvider extends ContentProvider

    //Defines a handle to the database helper object.
    private MainDatabaseHelper mOpenHelper;

    //Defines the database name
    private static final String DBNAME="mydb";

    //Holds the database object
    private SQLiteDatabase db;

    public boolean onCreate() {
```




```
        /*
         * Creates a new helper object. This method always returns quickly.
         * Notice that the database itself isn't created or opened
         * until SQLiteOpenHelper.getWritableDatabase is called
         */
        mOpenHelper=new SQLiteOpenHelper(
            getContext(),        //the application context
            DBNAME,              //the name of the database)
            null,                //uses the default SQLite cursor
            1                    //the version number
        );

        return true;
    }

    ...

    //Implements the provider's insert method
    public Cursor insert(Uri uri, ContentValues values) {
        //Insert code here to determine which table to open, handle error-
       checking, and so forth

        ...

        //Gets a writeable database.
        //This will trigger its creation if it doesn't already exist.
        db=mOpenHelper.getWritableDatabase();
    }
}
```

以下代码实现 SQLiteOpenHelper.onCreate()：

```
...
//A string that defines the SQL statement for creating a table
private static final String SQL_CREATE_MAIN="CREATE TABLE "+
    "main "+                //Table's name
    "("+                    //The columns in the table
    " _ID INTEGER PRIMARY KEY, "+
    " WORD TEXT"
    " FREQUENCY INTEGER "+
    " LOCALE TEXT)";
...
```

```
/**
 * Helper class that actually creates and manages the provider's underlying
 data repository.
 */
protected static final class MainDatabaseHelper extends SQLiteOpenHelper {

    MainDatabaseHelper(Context context) {
        super(context, DBNAME, null, 1);
    }

    /*
     * Creates the data repository. This is called when the provider attempts
       to open the
     * repository and SQLite reports that it doesn't exist.
     */
    public void onCreate(SQLiteDatabase db) {

        //Creates the main table
        db.execSQL(SQL_CREATE_MAIN);
    }
}
```

通过这个例子，可以看出 ContentProvider.onCreate() 和 SQLiteOpenHelper.onCreate()的相互调用。在实现 ContentProvider 的抽象方法时，除了 onCreate()外，需要考虑线程安全。

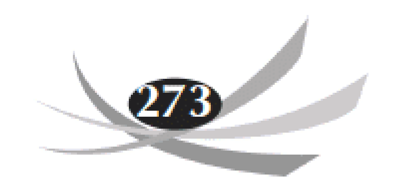
7. 定义访问权限

针对不同类型的存储方式，Android 系统的存储安全和有效的权限的要点有下面几个。

- 默认情况下，在内部存储的数据文件对其应用程序和 ContentProvider 是私有的。
- 外部存储的数据文件是公开的。无法使用 ContentProvider 限制其他程序访问外部存储中的文件，其他应用程序可以使用 API 读写它们。
- 在内部存储的文件或是 SQLite 数据库，可以由创建它的应用程序潜在地把读和写的权限付给其他的应用程序。如果要使用内部文件或数据库作为 ContentProvider 的数据源，必须在 Manifest 文件中设置权限，把其访问权限设为 world-readable 或 world-writeable，这些数据将不再受到保护。默认情况下内部存储器文件和数据库的访问权限是 private，对于 ContentProvider 也不应该改变。

如果要使用 ContentProvider 的权限来控制对数据的访问，就应该将数据存储在内部文件、SQLite 数据库，或“云”（例如，在远程服务器上）的数据中，并保持文件和数据库的 private 存储访问权限。

怎样进行 ContentProvider 的权限控制呢？



如果不做任何权限设置，所有的应用程序可以读取或写入 ContentProvider，即使这些数据的存储访问权限是私有的，因为默认情况下 ContentProvider 没有权限集。ContentProvider 的权限集需要在 Manifest 中，使用 ContentProvider 的属性和子元素来设置。在这里可以设置应用于整个 ContentProvider、特定的表、单一的记录或满足某些条件权限。

在 Manifest 文件中，可以使用 <permission>元素为 ContentProvider 定义一个或多个访问权限。为了这些权限的唯一性，可以用 Java 包名的方式定义 android:name 属性。例如，指定读权限的 com.example.app.provider.permission.READ_PROVIDER。

下面是 ContentProvider 数据范围由大到小的权限设置：

- 读写 ContentProvider 级别的权限：控制整个 ContentProvider 的读写权限，使用<provider>元素的 android:permission 属性指定。
- 读或写 ContentProvider 级别的权限：控制整个 ContentProvider 的读权限或写权限。使用<provider>元素的 android:readPermission 属性或 android:writePermission 属性来设定。
- 路径级别权限：对于 ContentProvider 中的 Content URI 指定数据的读、写或读/写权限。使用<provider>的子元素<path-permission>来指定要控制的每个 URI 数据的访问权限。在权限设置时，可以针对每个 Content URI，指定一个读/写权限、读权限或写权限，或所有三个权限。其中读取和写入权限覆盖读/写权限，并且路径级别的权限覆盖 Provider 级别的权限。
- 临时权限：对于临时访问的应用程序指定的权限。这个权限也适用于通常没有所需权限的应用程序。临时访问功能减少了应用程序在其 Manifest 中请求权限的数量。使用元素的 android:grantUriPermissions 属性，或者<provider>的子元素<grant-uri-permission>来设置临时权限。如果使用临时权限，无论什么时候从 Provider 中删除 Content URI 数据，都必须调用 Context.revokeUriPermission()方法，因为这个 Content URI 拥有临时权限。如果 android:grantUriPermissions 属性没有设置，默认就是 false。

与 Activity 和 Service 组件一样，ContentProvider 的子类也必须在其应用程序的 Manifest 文件中进行声明，才能够在系统中起作用。在 Manifest 文件中声明 ContentProvider 的元素为<provider>，使用 android:name 说明 ContentProvider 的名称。除了前面所提到的访问权限的属性和元素，还包括其他属性。

8.3.2 设计实例

上面介绍了 ContentProvider 设计和创建的方法，下面是一个 ContentProvider 的实例。

1. 创建 ContentProvider

为了更好地理解设计 ContentProvider 的原则，我们用一个完整的例子来讲解创建一个 Provider 的过程。

这个例子的 ContentProvider 创建，是基于第 7 章的 SOLite 学生信息数据库，这个数据库包括两张表 Students 和 Departments，这两个表之间有外键约束。下面按照前一节的设计过程，逐步进行完成 ContentProvider 的创建。

(1) 选择储存结构。

选择 SQLite 数据库。

(2) 定义资源标识符。

根据第 7 章 SQLite 数据库中定义的数据表，和自己定义的前缀定义所要创建的 Provider 的 Content URI 如下：

```
content://com.pinecone.technology.studentprovider/students
content://com.pinecone.technology.studentprovider/departments
```

这个定义在 Contract 类中实现。

(3) 定义资源标识符模式。

在 ContentProvider 类内代码的第一部分定义资源标识符模式，前一部分是常量定义，后一部分模式定义见代码 8.11。

代码 8.11　定义资源标识符模式

```
private static final String STUDENT_TABLE="students";
    private static final String DEPARTMENT_TABLE="departments";

    private static final int STUDENT=1;
    private static final int STUDENT_ID=2;
    private static final int DEPARTMENT=3;
    private static final int DEPARTMENT_ID=4;
    private static final UriMatcher MATCHER;

    static {
        MATCHER=new UriMatcher(UriMatcher.NO_MATCH);
        MATCHER.addURI(StudentsContract.AUTHORITY, "student", STUDENT);
        MATCHER.addURI(StudentsContract.AUTHORITY, "student/#", STUDENT_
        ID);
        MATCHER.addURI(StudentsContract.AUTHORITY, "department",
        DEPARTMENT);
        MATCHER.addURI(StudentsContract.AUTHORITY, "department/#",
                DEPARTMENT_ID);
}
```

(4) 定义 Contract 类。

根据 ContentProvider 的需要，定义一些常量，例如 Content URI、表名和列名，见代码 8.12。

代码 8.12　StudentsContract.java

```
public class StudentsContract {
    public static final String AUTHORITY="com.pinecone.technology.
    studentprovider";

    private StudentsContract() {
    }

    //inner class describing columns and their types
    public static final class Student implements BaseColumns {
        public static final Uri CONTENT_URI=Uri.parse("content://"
                +AUTHORITY+"/students");
        //Expose a content URI for this provider. This URI will be used to
        //access the ContentProvider
        //from within application components using a ContentResolver
        public static final String CONTENT_TYPE="vnd.android.cursor.dir/
        student";
        public static final String CONTENT_ITEM_TYPE="vnd.android.cursor.
        item/student";
        /**
         * SQL table columns
         */
        public static final String DEFAULT_STUDENT_SORT_ORDER="DeptId";
        public static final String NAME="StdName";
        public static final String AGE="Age";
        public static final String DEPT="DeptId";

    }

    public static final class Department implements BaseColumns {
        public static final Uri CONTENT_URI=Uri.parse("content://"
                +AUTHORITY+"/departments");
        //Expose a content URI for this provider. This URI will be used to
        //access the ContentProvider
        //from within application components using a ContentResolver
        public static final String CONTENT_TYPE="vnd.android.cursor.dir/
        department";
        public static final String CONTENT_ITEM_TYPE="vnd.android.cursor.
        item/department";
        /**
         * SQL table columns
         */
        public static final String NAME="DeptName";
```

```
        public static final String DEFAULT_DEPARTMENT_SORT_ORDER=
        "DeptName";
    }
}
```

(5) 定义 MIME 类型。对应这个例子的两个表，定义四个 MIME 类型：

```
vnd.android.cursor.dir/student
Vnd.android.cursor.item/student
vnd.android.cursor.dir/department
Vnd.android.cursor.item/department
```

这四个 MIME 类型在 StudentsContract 类中定义为常量(见代码 8.12)。

(6) 定义 ContentProvider。

定义 ContentProvider 的子类 StudentsProvider，首先定义资源标识符模式，然后具体实现 ContentProvider 的六个抽象方法 onCreate()、insert()、update()、delete()、query() 和 getType()(见代码 8.13)。

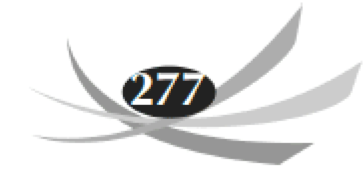
代码 8.13　StudentsProvider.java

```
public class StudentsProvider extends ContentProvider {

    private static final String STUDENT_TABLE="students";
    private static final String DEPARTMENT_TABLE="departments";

    private static final int STUDENT=1;
    private static final int STUDENT_ID=2;
    private static final int DEPARTMENT=3;
    private static final int DEPARTMENT_ID=4;
    private static final UriMatcher MATCHER;
    private DatabaseHelper dbHelper=null;

    static {
        MATCHER=new UriMatcher(UriMatcher.NO_MATCH);
        MATCHER.addURI(StudentsContract.AUTHORITY, "student", STUDENT);
        MATCHER.addURI(StudentsContract.AUTHORITY, "student/#", STUDENT_
        ID);
        MATCHER.addURI(StudentsContract.AUTHORITY, "department",
        DEPARTMENT);
        MATCHER.addURI(StudentsContract.AUTHORITY, "department/#",
                DEPARTMENT_ID);
    }
    private static HashMap<String, String> sStudentProjectionMap;
    static {
```




```
        sStudentProjectionMap=new HashMap<String, String>();
        sStudentProjectionMap.put(StudentsContract.Student._ID,
                StudentsContract.Student._ID);

        sStudentProjectionMap.put(StudentsContract.Student.NAME,
                StudentsContract.Student.NAME);
        sStudentProjectionMap.put(StudentsContract.Student.AGE,
                StudentsContract.Student.AGE);
        sStudentProjectionMap.put(StudentsContract.Department.NAME,
                StudentsContract.Department.NAME);
    }

    @Override
    public boolean onCreate() {
        Log.d("Provider", "onCreate");
        dbHelper=new DatabaseHelper(getContext());
        return ((dbHelper==null) ? false : true);
    }

    /**
     * Return the MIME type of the data at the given URI. This should
       start with
     * "vnd.android.cursor.item" for a single record, or
     * "vnd.android.cursor.dir" for multiple items. This method can
       be called
     * from multiple threads, as described in
     */
    @Override
    public String getType(Uri url) {
        final int match=MATCHER.match(url);
        switch(match) {
        case STUDENT:
            return StudentsContract.Student.CONTENT_TYPE;
        case STUDENT_ID:
            return StudentsContract.Student.CONTENT_ITEM_TYPE;
        case DEPARTMENT:
            return StudentsContract.Department.CONTENT_TYPE;
        case DEPARTMENT_ID:
            return StudentsContract.Department.CONTENT_ITEM_TYPE;
        default:
            throw new IllegalArgumentException("Unsupported URI: "+url);
        }
    }
```

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
        String[] selectionArgs, String sort) {
    Log.d("Provider", "query");
    SQLiteQueryBuilder qb=new SQLiteQueryBuilder();
    Cursor c=null;
    String orderBy=null;
    switch(MATCHER.match(url)) {
    case STUDENT:
        qb.setTables(STUDENT_TABLE);
        qb.setProjectionMap(sStudentProjectionMap);
        if(TextUtils.isEmpty(sort)) {
            orderBy=StudentsContract.Student.DEFAULT_STUDENT_SORT_ORDER;
        } else {
            orderBy=sort;
        }
        break;
    case STUDENT_ID:
        qb.setTables(STUDENT_TABLE);
        qb.setProjectionMap(sStudentProjectionMap);
        qb.appendWhere(StudentsContract.Student._ID+"="
                +url.getPathSegments().get(1));
        break;
    case DEPARTMENT:
        qb.setTables(DEPARTMENT_TABLE);
        qb.setProjectionMap(sStudentProjectionMap);
        if(TextUtils.isEmpty(sort)) {
            orderBy=StudentsContract.Department.DEFAULT_DEPARTMENT_
            SORT_ORDER;
        } else {
            orderBy=sort;
        }

        break;
    case DEPARTMENT_ID:
        qb.setTables(DEPARTMENT_TABLE);
        qb.setProjectionMap(sStudentProjectionMap);
        qb.appendWhere(StudentsContract.Department._ID+"="
                +url.getPathSegments().get(1));
        break;
    }
```




```
        c=qb.query(dbHelper.getReadableDatabase(), projection, selection,
                selectionArgs, null, null, orderBy);
        c.setNotificationUri(getContext().getContentResolver(), url);
        return (c);
    }

    @Override
    public Uri insert(Uri url, ContentValues initialValues) {
        if(MATCHER.match(url) !=STUDENT) {
            throw new IllegalArgumentException("Unknown URI "+url);
        }
        if(initialValues.containsKey(StudentsContract.Student.NAME)==
        false) {
            throw new SQLException(
                    "Failed to insert row because Book Name is needed "+url);
        }
        long rowID=dbHelper.getWritableDatabase().insert(STUDENT_TABLE,
                StudentsContract.Student.NAME, initialValues);
        if(rowID>0) {
            Uri uri=ContentUris.withAppendedId(
                    StudentsContract.Student.CONTENT_URI, rowID);
            getContext().getContentResolver().notifyChange(uri, null);
            return (uri);
        }
        throw new SQLException("Failed to insert row into "+url);
    }

    @Override
    public int delete(Uri url, String where, String[] whereArgs) {
        SQLiteDatabase db=dbHelper.getWritableDatabase();
        int count;
        switch(MATCHER.match(url)) {
        case STUDENT:
            count=db.delete(STUDENT_TABLE, where, whereArgs);
            break;
        case STUDENT_ID:
            String rowId=url.getPathSegments().get(1);
            count=db.delete(
                    STUDENT_TABLE,
                    StudentsContract.Student._ID
                            +"="
                            +rowId
                            +(!TextUtils.isEmpty(where) ? " AND ("+where
                                    +')' : ""), whereArgs);
```

```
            break;
        default:
            throw new IllegalArgumentException("Unknown URI "+url);
        }
        getContext().getContentResolver().notifyChange(url, null);
        return (count);
    }

    @Override
    public int update(Uri url, ContentValues values, String where,
            String[] whereArgs) {
        SQLiteDatabase db=dbHelper.getWritableDatabase();
        int count;
        switch(MATCHER.match(url)) {
        case STUDENT:
            count=db.update(STUDENT_TABLE, values, where, whereArgs);
            break;

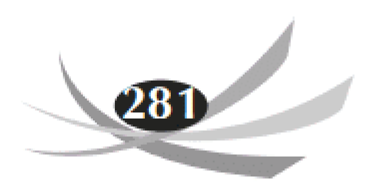
        case STUDENT_ID:
            String rowId=url.getPathSegments().get(1);
            count=db.update(
                    STUDENT_TABLE,
                    values,
                    StudentsContract.Student._ID
                            +"="
                            +rowId
                            +(!TextUtils.isEmpty(where) ? " AND ("+where
                                    +')' : ""), whereArgs);
            break;

        default:
            throw new IllegalArgumentException("Unknown URI "+url);
        }
          count = dbHelper.getWritableDatabase().update(STUDENT_TABLE,
          values, where, whereArgs);
        getContext().getContentResolver().notifyChange(url, null);
        return (count);
    }

}
```

（7）定义访问权限。

ContentProvider 的访问权限在 Manifest 文件中声明<provider>时定义。默认状



态下，所有的其他应用程序都可以访问这个 ContentProvider。

2. 注册 ContentProvider

最后要在 Manifest 文件中注册 ContentProvider：

```
<provider
    android:name=".StudentsProvider "
    android:authorities="com.androidbook.provider.StudentsProvider " />

}
```

8.4　实现数据加载

在前面的章节中，查看数据库的查询结果或 Provider 提供的数据时，每次查询的结果并不确定，需要实现数据的动态加载。Android 针对这一类的数据提供了一个机制，叫数据绑定。通过数据绑定，可以把动态的数据与称为 AdapterView 的图形控件连接起来，并自动根据数据的内容进行布局调整，按照某种规则显示给用户。在数据源和 AdapterView 之间起连接作用的类，在 Android 系统中称为适配器(Adapter)。

当想用合适的方式显示并操作一些数据(如数组，链表，数据库等)时，可以使用提供 Android 适配器的视图(AdapterView)，这种方式叫数据绑定(见图 8.2)。

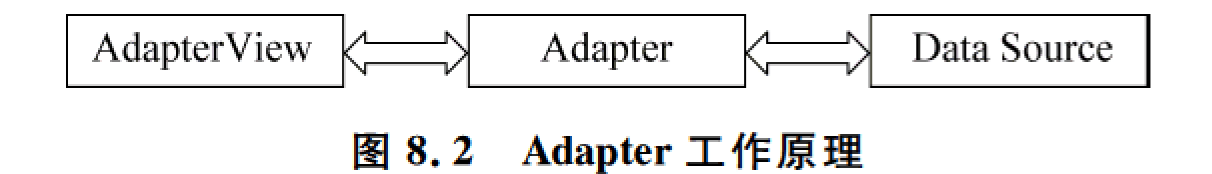


图 8.2　Adapter 工作原理

适配器就相当于一个通道，加了一些规则的通道，它可以使得流过通道的数据按照某种规则呈现出来。适配器是数据与数据显示控件(如 ListView、Gallery、Spinner)之间的桥梁，用来将数据绑定到显示控件上进行显示。例如，USB 是一个适配器，它有一些读取数据的规则，如果插入鼠标，则系统会通过 USB 获取的信息识别其是鼠标，系统可以对鼠标的操作做出反应；如果是 U 盘，系统会通过 USB 获取的信息识别其是 U 盘，可以对它进行信息存取操作。这里的 USB 就相当于适配器，鼠标或 U 盘的信息，就是系统通过适配器获取的数据。同样的道理，通过 Android 适配器的作用，Android 系统会识别出是数组还是数据库的数据，并根据适配器传递的信息做出合适的显示。

Android 提供多种适配器，开发时可以针对数据源的不同采用最方便的适配器，也可以自定义适配器完成复杂功能。使用这种机制，就可以把前面 Provider 的数据从用户界面上显示出来了。

下面首先介绍数据绑定的基本原理，然后介绍适合数据库数据显示的 ListView 图形控件。

8.4.1　基本原理

AdapterView 是 ViewGroup 的子类，其中画廊(Gallery)、列表视图(ListView)、微调

框控件(Spinner)和网格视图(GridView)等都是适配器视图 AdapterView 子类的例子，用来绑定到特定类型的数据并以一定的方式显示。AdapterView 对象有两个主要责任：用数据填充布局和响应用户的选择事件。

常见的适配器有 SimpleAdapter、SimpleCursorAdapter、ArrayAdapter。从名称可以看出 ArrayAdapter 使用数组作为数据源，SimpleCursorAdapter 使用游标作为数据源，而 SimpleAdapter 将一个 List 作为数据源，可以让 ListView 进行更加个性化的显示。

下面使用 Android 下拉菜单 Spinner 控件，来举例说明数据绑定的机制。

(1) 设置应用的布局文件。

首先设计用户图形界面的布局文件，把显示数据结果的 Spinner 控件作为界面中的一个组件(见代码 8.14)。

代码 8.14 布局文件 c07_spinner.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dip"
        android:text="@string/planet_prompt" />

    <Spinner
        android:id="@+id/spinner"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:drawSelectorOnTop="true"
        android:prompt="@string/planet_prompt" />

</LinearLayout>
```

(2) 在 Activity 中获得 Spinner 控件对象。

在定义用户界面的 Activity 子类中，通过布局文件代码 8.14 中所定义的 Spinner 的 ID，在 onCreate()方法中使用 findViewById()获取 Spinner 对象。代码如下：

```
setContentView(R.layout.c07_spinner);
Spinner s=(Spinner) findViewById(R.id.spinner);
```

(3) 实现 Spinner 与数据源的数据绑定。

获取 Spinner 对象后，在此 Activity 子类的 onCreate()方法内实现数据绑定。

首先为 Spinner 控件创建适配器，获得 arrays.xml 资源文件中数组 planets；接下来

将数据显示界面声明为 android. R. layout. simple_spinner_item 布局模式，将此适配器与控件对象绑定。这个模式是为 Spinner 类预定义好的布局模式。然后创建下拉菜单(见代码 8.15)。

代码 8.15 创建适配器

```
ArrayAdapter<CharSequence>adapter=ArrayAdapter.createFromResource(this,
R.array.planets, android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown
_item);
s.setAdapter(adapter);
```

适配器 ArrayAdapter 是适用于数组数据的适配器，用外部数据创建一个适配器对象可以使用其 createFromResource()方法。其参数说明如下：

- Context context：应用程序的环境。
- int textArrayResId：作为数据源的数组。
- int textViewResId：用于创建视图的布局模式。

创建下拉菜单使用 ArrayAdapter 的 setDropDownViewResource()方法，其参数为 int resource，意即布局资源定义的下拉菜单视图。

(4) 定义数据源数组。

在资源文件 arrays. xml 文件中定义数组 planets，并设置数组值(见代码 8.16)。

代码 8.16 arrays. xml

```
<resources>
<string name="app_name">Spinner</string>
<string-array name="planets">
<item>Mercury</item>
<item>Venus</item>
<item>Earth</item>
<item>Mars</item>
<item>Jupiter</item>
<item>Saturn</item>
<item>Uranus</item>
<item>Neptune</item>
<item>Pluto</item>
</string-array>
<string name="planet_prompt">Select a planet</string>
</resources>
```

运行这个例子程序，就得到图 8.3 显示的界面。这里的数据源，也就是 arrays 中定义的 planets 数组，通过 ArrayAdapter 这个适配器，与 Spinner 这个图形显示控件联系起来，使数组数据直接按照列表的形式显示，不必再做布局的设计。这个功能就是填充布局的作用。



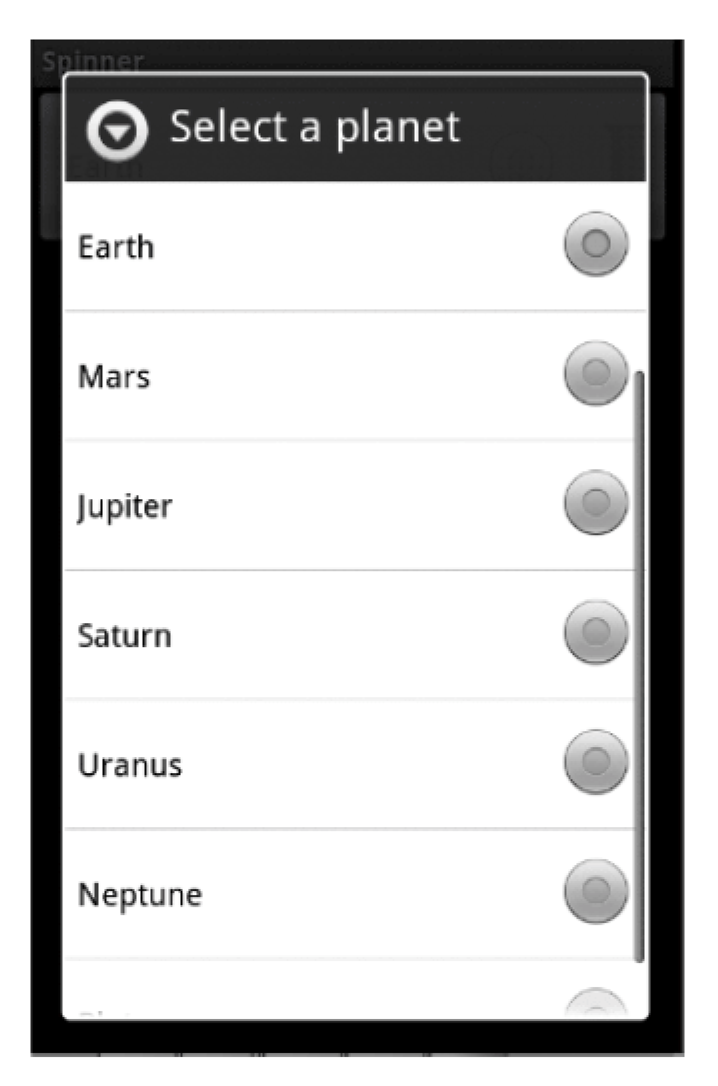


图 8.3　通过数组给列表赋值

如果布局是动态的或者非预定义的，可以在运行时使用一个布局子类 AdapterView 来填充布局。AdapterView 类的子类使用一个适配器将数据绑定到它的布局。适配器把数据源和 AdapterView 布局之间连接起来，适配器检索数据，例如把数据从数组或者数据库提取出来，将其转换成可以添加到 AdapterView 布局视图中的条目。

通用的适配器布局包括 ListView（图 8.4 左图）和 Grid View（图 8.4 右图）。

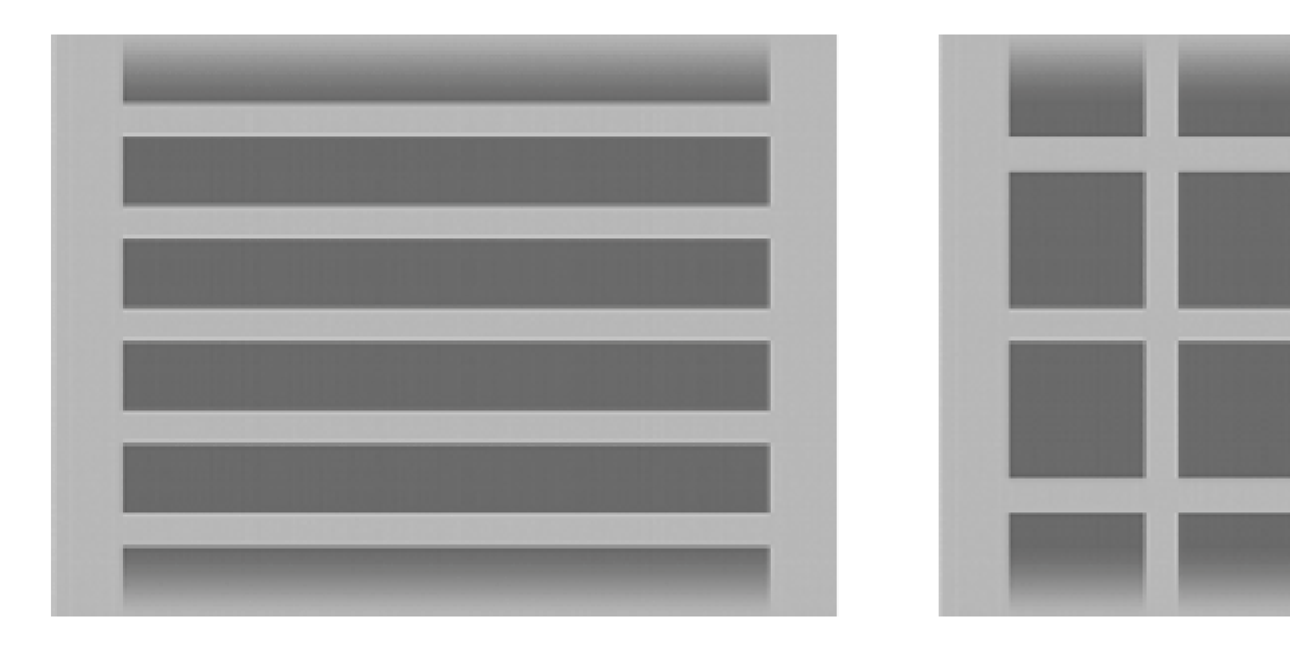


图 8.4　ListView 和 GridView 显示效果

8.4.2　ListView 控件

ListView 是 AdapterView 的子类，用于列表显示。ListView 的定义有两种方式：

- 继承 ListActivity 类，使用其内置的 ListView 对象。
- 在布局文件中定义自定义视图的 ListView。

ListActivity 是一个专门显示 ListView 的 Activity 类，它内置了 ListView 对象，只要设置了数据源，就会自动地显示出来。虽然 ListActivity 内置了 ListView 对象，但依然可以在布局文件中自定义视图。

自定义视图时，在布局文件中要注意设置 ListView 对象的 id 为"@id/android:list "；而在 Java 代码里使用 android. R. id. list 来引用 ListView 视图。在使用 ListActivity 来显示 ListView 视图时，如果使用了自定义的布局文件，通过 setContentView()方法进行绑定；如果不使用自定义的布局文件，这个步骤可以省略。

Android 系统提供了多种模板进行选择，例如：

- Simple_list_item_1 表示每行有一个 TextView；
- Simple_list_item_2 表示每行有两个 TextView；
- Simple_list_item_checked 表示每行带 CheckView 的项；
- Simple_list_item_multiple_choise 表示每行有一个 TextView 并可以多选；
- Simple_list_item_single_choice 表示每行有一个 TextView，但只能进行单选。

如果以上模板还无法满足要求，那只能自定义模板。

自定义模板可以根据自己的需要定义成任意的格式，包括图片、方案及其他可显示的视图，而且还要考虑怎样进行视图的数据绑定。

ListView 是一个经常用到的控件，ListView 里面的每个子项 Item 可以是一个字符串，也可以是一个组合控件。ListView 要正常显示需要三个元素：

- 用来显示数据的 ListView 控件；
- 用来显示的数据；
- 用来将数据和 ListView 绑定的 ListAdapter。

对 ListView 进行数据绑定，必须选择使用适配器。其中最常与 ListView 进行配合使用的有 ArrayAdapter、CursorAdapter 及 SimpleAdapter 等。

下面这个例子，说明了如何使用 ListView 显示 8. 3. 2 节所创建的 ContentProvider 的内容(见代码 8. 17 和 8. 18)。

代码 8. 17　StudentsContract. java

```
public class StudentsContract {
    public static final String AUTHORITY="com.pinecone.technology.
    studentprovider";

    private StudentsContract() {
    }

    //inner class describing columns and their types
    public static final class Student implements BaseColumns {
        public static final Uri CONTENT_URI=Uri.parse("content://"
                +AUTHORITY+"/students");
        //Expose a content URI for this provider. This URI will be used to
        //access the ContentProvider
        //from within application components using a ContentResolver
        public static final String CONTENT_TYPE="vnd.android.cursor.dir/
        student";
        public static final String CONTENT_ITEM_TYPE="vnd.android.cursor.
        item/student";
```

```
        /**
         * SQL table columns
         */
        public static final String DEFAULT_STUDENT_SORT_ORDER="DeptId";
        public static final String NAME="StdName";
        public static final String AGE="Age";
        public static final String DEPT="DeptId";

    }

    public static final class Department implements BaseColumns {
        public static final Uri CONTENT_URI=Uri.parse("content://"
                +AUTHORITY+"/departments");
        //Expose a content URI for this provider. This URI will be used to
        //access the ContentProvider
        //from within application components using a ContentResolver
        public static final String CONTENT_TYPE="vnd.android.cursor.dir/
        department";
        public static final String CONTENT_ITEM_TYPE="vnd.android.cursor.
        item/department";
        /**
         * SQL table columns
         */
        public static final String NAME="DeptName";
        public static final String DEFAULT_DEPARTMENT_SORT_ORDER=
        "DeptName";
    }
}
```

代码 8.18　MainActivity.java

```
public class MainActivity extends ListActivity {

    private static final String TAG="MainActivity";
    private Cursor mCurser;
    private SimpleCursorAdapter mCursorAdapter;
    private ContentValues values;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        values=new ContentValues();
        values.put(StudentsContract.Student.NAME, "学生 1");
        values.put(StudentsContract.Student.AGE, "30");
```

```
        values.put(StudentsContract.Student.DEPT, "3");
        getContentResolver().insert(StudentsContract.Student.CONTENT_URI,
                values);
        String[] mProjection={ StudentsContract.Student._ID,
                StudentsContract.Student.NAME, StudentsContract.Student.
                AGE };
        String mSelectionClause=null;

        //Initializes an array to contain selection arguments
        String[] mSelectionArgs=null;
        mCurser=getContentResolver().query(
                StudentsContract.Student.CONTENT_URI, mProjection,
                mSelectionClause, mSelectionArgs, null);
        if(null==mCurser) {
            Log.i(TAG, "Curser is null");
        } else {
            mCursorAdapter=new SimpleCursorAdapter(this,
                    android.R.layout.simple_list_item_2, mCurser, new
                    String[] {
                            StudentsContract.Student.NAME,
                            StudentsContract.Student.AGE }, new int[] {
                            android.R.id.text1, android.R.id.text2 }, 0);
        }
        setListAdapter(mCursorAdapter);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        //Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

8.5 本章小结

本章主要介绍了Android的四大基础组件之一ContentProvider组件。ContentProvider是Android系统提供给用户的一个接口,用于管理如何访问应用程序私有数据的存储库。这里的数据包括结构化存储数据和非结构化存储数据。

应用程序使用ContentResolver类客户端对象来访问ContentProvider的数据,可以

称其为访问提供器。ContentResolver 的方法提供了基本的 CRUD(创建,检索,更新和删除)数据存储的功能。

应用程序通过 ContentResolver 的对象访问 Provide 时,所使用的方法会调用 ContentProvider 一个具体子类对象的相同名字的方法。ContentProvider 对象通过 URI 来选择要访问 Provider 的表和数据,Content URI 就是 ContentProvider 中数据的内容统一资源标识,能够在存储介质中唯一标识 ContentProvider 中数据所在的具体位置。

当想用合适的方式显示并操作一些数据(如数组、链表、数据库等)时,可以使用 AdapterView 来显示交互界面,这种方式叫数据绑定。ContentProvider 所提供的数据可以使用数据绑定的方式来显示。

第 9 章 触摸事件处理

触摸屏是智能手机和平板电脑最重要的输入输出工具，用户在与系统或应用程序交互过程中，大多数操作都是通过触摸屏来完成的。触摸屏由特殊材料制成，可以获取屏幕上的压力，并转换成屏幕坐标。这些信息可以被转换成数据，并被传递到软件里。所以，应用程序需要经常处理用户的触摸输入，包括一个手指的触摸和多个手指的触摸。

9.1 理解触摸事件

触摸屏可以感知手指的触摸压力，识别手指是抬起、按下或者是移动；而且可以将触点转换成屏幕坐标，通过计算屏幕坐标的变化，可以识别触点的移动方式。当用户触摸屏幕时，触摸事件就会产生。

1. 手势

手势是指系统可识别的，用户在对屏幕显示对象操作时，手指在触摸屏上抬起、按下或移动的方式。在 Android 系统中支持的核心手势包括以下几种（见图 9.1）。

(1) 触摸(Touch)：按下，抬起。触发项目的默认操作。

(2) 长按(Long Press)：按下，等待，抬起。进入选择模式，使用户可以选择视图中的单个或者多个项目，并选择上下文操作栏的功能。

(3) 滑动(Swipe)：按下，移动，抬起。滚动内容或者在同一层级的不同视图间切换。

(4) 拖曳(Drag)：长按，移动，抬起。重新排列视图中的数据或者将数据移动到容器（例如主屏幕上的目录）中。

(5) 双击(Double Touch)：快速两次触摸。放大内容。在文字选择中作为辅助手势。

(6) 放大(Pinch Open)：用两个手指按住，向相互远离的方向移动，抬起。放大内容。

(7) 缩小(Pinch Close)：用两个手指按住，向相互接近的方向移动，抬起。缩小内容。

2. 触摸事件 MotionEvent

在 Android 系统中，触摸事件由 MotionEvent 类来描述。产生一个触摸事件，系统就会创建一个 MotionEvent 对象，该对象包含了触摸事件发生的时间和位置，以及发生触摸事件所在区域的压力、大小和方向。在应用中，MotionEvent 对象会被传递到某些方法中，其中包括 View 类的 onTouchEvent()方法。因为 View 类是很多控件的父类，这就意味着很多控件都可以通过 MotionEvent 与用户进行交换。例如，MapView 控件可以接受










图 9.1　手势种类

触摸事件，允许用户水平移动地图到感兴趣的地方；或者虚拟键盘对象接收触摸事件激活虚拟键，实现在界面中输入文本。

从用户手指触摸设备屏幕开始，到手指离开设备屏幕结束，Android 系统会产生一系列与手指运动相关的触摸事件，每个触摸事件都记录手指运动的信息，称这些触摸事件为一个事件序列。实际上，很多触摸屏设备可以同时记录多个手指的运动轨迹，这样每个运动轨迹都会产生一个触摸事件序列。每个序列是从用户触摸屏幕开始，当用户在屏幕上移动时，这个序列会持续添加；当手指从屏幕上抬起后，这个序列也就结束。

MotionEvent 类中定义了动作常量表示触摸事件的动作类型，主要包括 ACTION_DOWN、ACTION_UP、ACTION_CANCEL、ACTION_MOVE 等。当用户首次触摸屏幕时，系统会将带有 ACTION_DOWN 的触摸事件传递给相应的视图控件；当手指在屏幕上移动是 ACTION_MOVE；当抬起手指是 ACTION_UP；而且在手指抬起之前系统可能会产生很多 ACTION_MOVE 的触摸事件。所有这些触摸事件都会产生相应的 MotionEvent 对象，其中包含了动作的种类、触摸发生的位置、触摸的压力、触摸的面积、动作发生的时间和初始 ACTION_DOWN 的时间等属性。而 ACTION_OUTSIDE 是一个特殊动作，是指当手指移动到在窗体之外时，系统仍然可以得到这样的触控事件。一般来说，一个手指触摸屏幕会触发一个最简单的触摸事件序列首先应该是 ACTION_DOWN，然后会有多个 ACTION_MOVE，最后一个 ACTION_UP 结束。

有些触摸屏设备可以在同一时间发现多个移动轨迹，称为多点触控。Android 系统的 MotionEvent 类也支持多点触控。其中包括 ACTION_POINTER_DOWN 和 ACTION_POINTER_UP 两个动作常量，分别表示除第一个之外的手指在屏幕触摸时的动作和除第一个之外的手指从屏幕抬起时的动作。

MotionEvent 类使用两个属性来表示具体的触点，一个是触点 ID，一个是触点索引，并提供许多方法用来查询每个触点位置以及其他属性，如 getX(int)、getY(int)、getAxisValue(int)、getPointerId(int)、getToolType(int)等。这些方法中大部分接收触点索引作为参数而不是触点 ID。getPointerCount()方法可以获得指针总数量，而触点索引的取值范围是从 0 开始，到指针总数量减 1 结束。当发生多点触摸事件时，每个触点索引是会发生变化的，而触点 ID 在触摸点移动过程中不会发生变化。getPointerId(int)方法传入触点索引可以获得触点 ID，而 findPointerIndex(int)方法传入触点 ID 可以获得触点索引。

当多点触控事件发生时，系统可能产生的事件见表 9.1。

表 9.1　触控事件

触摸事件	描　述
MotionEvent. ACTION_DOWN	新的触摸事件
MotionEvent. ACTION_MOVE	移动手指
MotionEvent. ACTION_UP	手指抬起
MotionEvent. ACTION_CANCEL	删除事件
MotionEvent. ACTION_POINTER_DOWN	多点按下
MotionEvent. ACTION_POINTER_UP	多点抬起

由于应用程序的开发，在真机调试之前都是在模拟机上调试的，在处理多点触控事件时，需要注意模拟机和真机上触屏事件有一些不同。

(1) 屏幕的精度。

在模拟机上的精度是整数，例如 52×20；而在真机上有小数，例如 42.8374×25.293747。MotionEvent 的位置由 X 轴和 Y 轴坐标组成的，X 轴表示从视图左手边到触屏点的距离，Y 轴表示从视图的顶端到触屏点的距离。

(2) 触摸事件中压力描述。

在模拟机上的压力值为 0，而在真机上输出的压力值表示手指在触摸屏上向下的力度。如果使用小拇指的指尖轻轻触摸，则触摸事件的压力和大小比较小；如果用力使用大拇指，则触摸事件的压力和大小都是比较大的。触摸事件的压力和大小是在 0～1。但是，对于不同的设备来说，没有一个绝对的值用来比较触摸事件的压力和尺寸的大小；而对于同一个设备来说，只能相对地比较触摸事件之间的压力和尺寸大小，不能拿一个绝对的值来确定压力和尺寸的大小。例如，在某些设备上这个值从来不超过 0.8，而某些设备上这个值从来不超过 0.2。

（3）触摸事件序列。

在 Android 中使用触摸屏时，如果应用程序在模拟器中运行，当鼠标单击一次模拟器屏然后释放后，会先触发 ACTION_DOWN 然后是 ACTION_UP 共两个触摸事件，只有在屏幕上移动时才会触发 ACTION_MOVE 的动作；但在真机中测试时，单击会首先产生 ACTION_DOWN 触摸事件，如果手指不抬起即使不移动也会一直产生 ACTION_MOVE 触摸事件，手指只有离开屏幕时才会产生 ACTION_UP 触摸事件。

当手指触摸到设备的边界位置时，触摸事件的边界标记会被检测到。而 Android 文档上显示，当触摸到设备的边界（顶、底、左和右）时，这个标记会被设置。但是有时 getEdgeFlags()方法总是显示为 0。实际上，有些硬件很难检测到触摸在设备的边界上，所以 Android 系统不可能设置这个参数。MotionEvent 类提供了 setEdgeFlags()方法，目的是可以自己来设置这个值。

9.2 事件传递机制

当触摸事件发生时，MotionEvent 对象会作为参数被传递到应用程序中的相应方法中。在 Android 系统中，ViewGroup、View、Activity 及其子类都提供了处理触摸事件的回调方法。触摸事件与前面所接触的按钮、编辑框、菜单和动作条等图形控件的事件有所不同。当用户直接针对这些图形控件进行键盘操作时，应用程序直接回调对应的注册监听器进行处理。但当触摸事件发生时，会遇到多种情况。例如屏幕中包含一个 ViewGroup，而这个 ViewGroup 又包含一个子 View 时，Android 系统如何处理触摸事件呢？到底是 ViewGroup 来处理触摸事件，还是子 View 来处理触摸事件呢？如果一个视图控件注册了 OnClickListener 和 OnLongClickListener 监听器，分别实现了 onClick()和 onLongClick()方法，当用户触摸到屏幕上的这个视图控件时，Android 系统如何区分？应当使用 onTouchEvent()方法，还是 onClick()方法，或是 onLongClick()方法来处理事件呢？

如果要解答这些问题，需要深入理解触摸事件的传递和消费机制。在 Android 系统中，同一个触摸事件可以按次序传递到不同的视图控件，所以可以依次被视图控件处理，如果某视图控件完全响应而且不再传递这个触摸事件，则称为消费了触摸事件。

真正理解了触摸事件的传递和消费机制，才能编写出正确响应界面操作的代码，尤其当屏幕上的不同视图控件，需要针对同一个界面触摸事件做出不同响应的时候。例如，应用程序在桌面上设置了一个控件，当用户针对控件做各种操作时，有时桌面本身要对用户的操作做出响应，有时忽略。只有搞清楚事件触发和传递的机制才有可能保证在界面布局非常复杂的情况下，UI 控件仍然能正确响应用户的操作。

在处理触摸事件时，一个用户的操作可能会被传递到不同的控件，或同一个控件的不同监听方法内进行处理。如果任何一个接收该事件的方法在处理完后返回了 true，则该事件就算处理完成了，其他的视图或者监听方法就不会再有机会处理该事件了。

9.2.1 内外层次之间

一般来说，用户界面的树形结构是由多个 View 和 ViewGroup 形成的，它们之间具

有由外向内的包含关系，而触摸事件可以在相邻的层次之间传递，传递方向先从外向内，然后从内向外。从外向内传递就是从最外层的根元素依次递归向其包含的子元素传递，一直到最内层子元素，或中间某个元素消费了触摸事件，结束了传递；从内向外就是从最内层子元素依次递归向外层传递，直到根元素或中间某个元素消费了触摸事件，结束了传递。

1. 处理触摸事件传递的方法

Android 系统使用下面三个方法来处理触摸事件的传递：

1）public boolean dispatchTouchEvent(MotionEvent ev)

这个方法是 View、Activity 和 ViewGroup 类中的方法，用来分发触摸事件。可以把这个方法作为一个控制器，由其决定如何处理路由触摸事件。View 类中的 dispatchTouchEvent()方法判断是将触摸事件传递给 View. OnTouchListener. onTouchEvent()或者 View. onTouchEvent()方法。ViewGroup 类中的 dispatchTouchEvent()方法覆盖了 View. dispatchTouchEvent()方法，其包含了更复杂的算法，用来弄清楚哪个子视图应该得到触摸事件，而且需要调用子视图的 dispatchTouchEvent()方法。

2）public boolean onInterceptTouchEvent(MotionEvent ev)

这个方法只是 ViewGroup 类中的方法，用来拦截触摸事件。ViewGroup 一般都会包含 View，而且 ViewGroup 和 View 都包含 onTouchEvent()方法。如果 onInterceptTouchEvent()返回 true 时，则执行此 ViewGroup 中的 onTouchEvent()方法；如果 onInterceptTouchEvent()返回 false 时，触摸事件将传递给 View，由 View 的 dispatchTouchEvent()方法再来开始这个事件的分发。

3）public boolean onTouchEvent(MotionEvent ev)

这个方法可以用来处理触摸事件。该方法在 View、ViewGroup 以及 Activity 类中都有定义，并且所有的 View 子类全部重写了该方法，包括 Layouts、Buttons、Lists、Surfaces、Clocks 等，这说明所有的这些组件都可以使用触摸事件进行交互，应用程序可以通过该方法处理手机屏幕的触摸事件。

2. 触摸事件传递方法的测试

下面一个例子，可以测试这种触摸事件的传递方法。在一个自定义的布局中包含自定义的 TextView 控件，并且分别覆盖 Activity、自定义布局和 TextView 中的三个方法，方法的内容主要是日志输出和控制触摸事件的传递。

首先，创建一个 LinearLayout 的子类，覆盖上面所列出的处理触摸事件的三个方法 dispatchTouchEvent()、onInterceptTouchEvent()和 onTouchEvent()，处理在这个布局上发生的触摸事件。当在不同的触摸事件发生时，根据事件的类型，输出带有事件处理方法和事件类型的日志信息（见代码 9.1）。

代码 9.1　自定义布局

```
public class MyLayoutView extends LinearLayout {
    private final String TAG="MyLayoutView";

    public MyLayoutView(Context context, AttributeSet attrs) {
```

```
        super(context, attrs);
        Log.d(TAG, TAG);
    }

    @Override
    public boolean dispatchTouchEvent(MotionEvent ev) {
        int action=ev.getAction();
        switch(action) {
        case MotionEvent.ACTION_DOWN:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_CANCEL");
            break;
        }
        return super.dispatchTouchEvent(ev);
    }

    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        int action=ev.getAction();
        switch(action) {
        case MotionEvent.ACTION_DOWN:
            Log.d(TAG, "onInterceptTouchEvent action:ACTION_DOWN");
            //return true;
            break;
        case MotionEvent.ACTION_MOVE:
            Log.d(TAG, "onInterceptTouchEvent action:ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.d(TAG, "onInterceptTouchEvent action:ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.d(TAG, "onInterceptTouchEvent action:ACTION_CANCEL");
            break;
        }
        return false;
    }
```

```
    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        int action=ev.getAction();
        switch(action) {
        case MotionEvent.ACTION_DOWN:
            Log.d(TAG, "onTouchEvent action:ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.d(TAG, "onTouchEvent action:ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.d(TAG, "onTouchEvent action:ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.d(TAG, "onTouchEvent action:ACTION_CANCEL");
            break;
        }
        return false;
    }
}
```

定义一个 TextView 类的子类作为用户界面布局中的输入框，并覆盖其触摸事件处理的方法 dispatchTouchEvent()和 onTouchEvent()，实现当在不同的触摸事件发生时，根据事件的类型，输出带有事件处理方法和事件类型的日志信息(见代码 9.2)。

代码 9.2 自定义输入框

```
public class MyTextView extends TextView {

    private final String TAG="MyTextView";

    public MyTextView(Context context, AttributeSet attrs) {
        super(context, attrs);
        Log.d(TAG, TAG);
    }

    @Override
    public boolean dispatchTouchEvent(MotionEvent ev) {
        int action=ev.getAction();
        switch(action) {
        case MotionEvent.ACTION_DOWN:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_DOWN");
            break;
```

```
        case MotionEvent.ACTION_MOVE:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.d(TAG, "dispatchTouchEvent action:ACTION_CANCEL");
            break;
        }
        return super.dispatchTouchEvent(ev);
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        int action=ev.getAction();
        switch(action) {
        case MotionEvent.ACTION_DOWN:
            Log.d(TAG, "onTouchEvent action:ACTION_DOWN");
            break;
        case MotionEvent.ACTION_MOVE:
            Log.d(TAG, "onTouchEvent action:ACTION_MOVE");
            break;
        case MotionEvent.ACTION_UP:
            Log.d(TAG, "onTouchEvent action:ACTION_UP");
            break;
        case MotionEvent.ACTION_CANCEL:
            Log.d(TAG, "onTouchEvent action:ACTION_CANCEL");
            break;
        }
        return true;
    }
}
```

然后，定义用于测试的 Activity 的布局文件，其中定义触摸事件传递的层次关系（见代码 9.3）。

代码 9.3　XML 布局文件

```
<?xml version="1.0" encoding="utf-8"?>
<cn.edu.uibe.mcommerce.chapter11.motionEvent.MyLayoutView xmlns:android
="http://schemas.android.com/apk/res/android"
    android:layout_width="300dip"
    android:layout_height="200dip"
    android:layout_gravity="center"
    android:background="#ff0000"
    android:gravity="center"
```

```
    android:orientation="vertical"
    android:tag="My Layout">

    <cn.edu.uibe.mcommerce.chapter11.motionEvent.motionEvent.MyTextView
        android:id="@+id/tv"
        android:layout_width="200dip"
        android:layout_height="100dip"
        android:background="#00FF00"
        android:gravity="center"
        android:text="My TextView"
        android:textColor="#0000FF"
        android:textSize="20sp"
        android:textStyle="bold" />

</cn.edu.uibe.mcommerce.chapter11.motionEvent.motionEvent.MyLayoutView>
```

定义好布局文件后，创建一个 Activity，导入所定义的 XML 布局文件，运行应用程序，可以显示出图 9.2 的界面。

图 9.2 显示的界面中，从外到内包括一个 Activity（View）、一个自定义 Layout（ViewGroup）和一个自定义的 TextView（View）。深色部分是自定义布局，浅色部分是自定义的 TextView。如果用手指触摸界面，就会产生一系列触摸事件。首先产生的是 MotionEvent. ACTION_DOWN 事件，这是触摸事件系列的第一个事件。

Touch Me

图 9.2　运行效果

在触摸事件传递过程中，ACTION_DOWN 经过的界面元素实际上都是继承了 View 或者 ViewGroup 类，这两个类中都包含 dispatchTouchEvent()方法，但是触摸事件处理的最外层却不是这些界面元素，而会首先从调用当前 Activity 的 dispatchTouchEvent()方法，然后才将触摸事件传递给其中的 View 或者 ViewGroup 元素。这样，触摸事件首先从最外层 View 或者 ViewGroup 元素向内层 View 或者 ViewGroup 元素传递。

在触摸事件从外层的界面元素向内层的界面元素的传递过程中，如果事件传递到继承了 ViewGroup 类的界面元素，则会调用 ViewGroup 类的 onInterceptTouchEvent()方法，这个方法表示是否拦截触摸事件。如果这个方法返回 true，表示这个 ViewGroup 拦截了事件的传递，触摸事件不会再往下传递给它的子 View 元素，而是由这个 ViewGroup 元素处理，调用其 onTouchEvent()方法；如果在传递的过程中没有 ViewGroup 拦截事件，即经过的所有 onInterceptTouchEvent()方法都返回 false，那么触摸事件最终会传递至最内层的界面元素，一般是一个视图控件，当然也可以是一个 ViewGroup 元素（其内部不包含任何元素）。

如果最后事件传递到一个 View 元素，而非 ViewGroup 元素，那么会首先调用这个 View 的 OnTouchListener 的 onTouch()方法或者调用 View 的 onTouchEvent()方法，其

默认返回 true；如果最后事件传递到一个 ViewGroup 元素，会调用它的 onTouchEvent()方法，其默认返回 false，这样就完成了触摸事件从外向内的传递。

在上面示例中的 Activity 和自定义布局的三种方法的返回值都为 false，则触摸事件传递到最内层的自定义 TextView。由于自定义的 TextView 的 onTouchEvent()方法返回值为 true，所以触摸事件被消费。图 9.3 示意了从外向内传递触摸事件的路径。

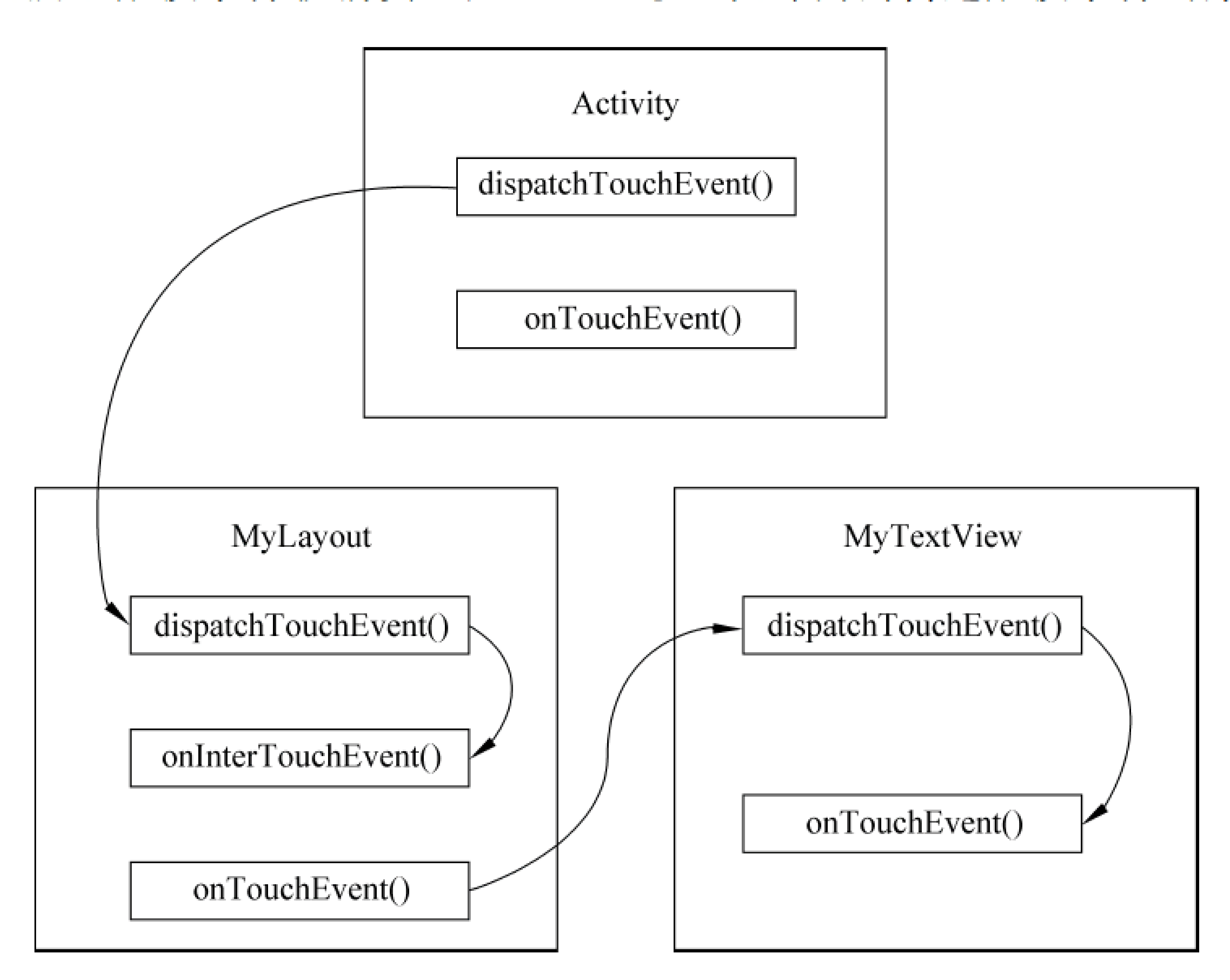
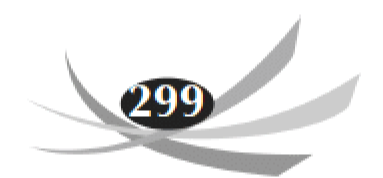


图 9.3　从外向内传递触摸事件的路径

下面是这个过程的日志输出：

```
07-16 21:43:07.809: D/InterceptTouchActivity(18468): dispatchTouchEvent
action:ACTION_DOWN
07-16 21:43:07.814: D/MyLayoutView(18468): dispatchTouchEvent action:
ACTION_DOWN
07-16 21:43:07.814: D/MyLayoutView(18468): onInterceptTouchEvent action:
ACTION_DOWN
07-16 21:43:07.814: D/MyTextView(18468): dispatchTouchEvent action:ACTION
_DOWN
07-16 21:43:07.814: D/MyTextView(18468): onTouchEvent action:ACTION_DOWN
07-16 21:43:07.834: D/InterceptTouchActivity(18468): dispatchTouchEvent
action:ACTION_MOVE
07-16 21:43:07.834: D/MyLayoutView(18468): dispatchTouchEvent action:
ACTION_MOVE
07-16 21:43:07.834: D/MyLayoutView(18468): onInterceptTouchEvent action:
ACTION_MOVE
07-16 21:43:07.834: D/MyTextView(18468): dispatchTouchEvent action:ACTION
_MOVE
```




```
07-16 21:43:07.834: D/MyTextView(18468): onTouchEvent action:ACTION_MOVE
07-16 21:43:07.834: D/InterceptTouchActivity(18468): dispatchTouchEvent
action:ACTION_UP
07-16 21:43:07.834: D/MyLayoutView(18468): dispatchTouchEvent action:
ACTION_UP
07-16 21:43:07.834: D/MyLayoutView(18468): onInterceptTouchEvent action:
ACTION_UP
07-16 21:43:07.834: D/MyTextView(18468): dispatchTouchEvent action:ACTION
_UP
07-16 21:43:07.834: D/MyTextView(18468): onTouchEvent action:ACTION_UP
```

如果上面例子中自定义 TextView 的 onTouchEvent()方法返回了 false，则接下来的触摸事件，即 ACTION_DOWN 的此事件系列的后续触摸事件，就不会再传递到这个 View 或者 ViewGroup 元素，而会在其父元素终止，并且调用其父元素的 onTouchEvent()方法。

这就是说，如果最内层界面元素的 onTouchEvent()方法返回了 false，则事件会自内向外再次传递，直到某个界面元素 onTouchEvent()方法返回 true。这时，此事件系列的后续触摸事件，也会直接由这个界面元素的 onTouchEvent()方法来处理。

如果从内向外的所有 View 或者 ViewGroup 界面元素的 onTouchEvent()方法都返回 false，那么 ACTION_DOWN 的此事件系列的后续触摸事件最后会由 Activity 处理，即调用 Activity 的 onTouchEvent()方法，并且最终结束触摸事件的传递，这就是从内向外的触摸事件的传递。

还是使用上面的例子，只是在代码 9.2 中，将自定义 TextView 的 onTouchEvent()方法返回值改为 false，触摸事件就从内向外传递触摸事件(见图 9.4 虚线所示)。

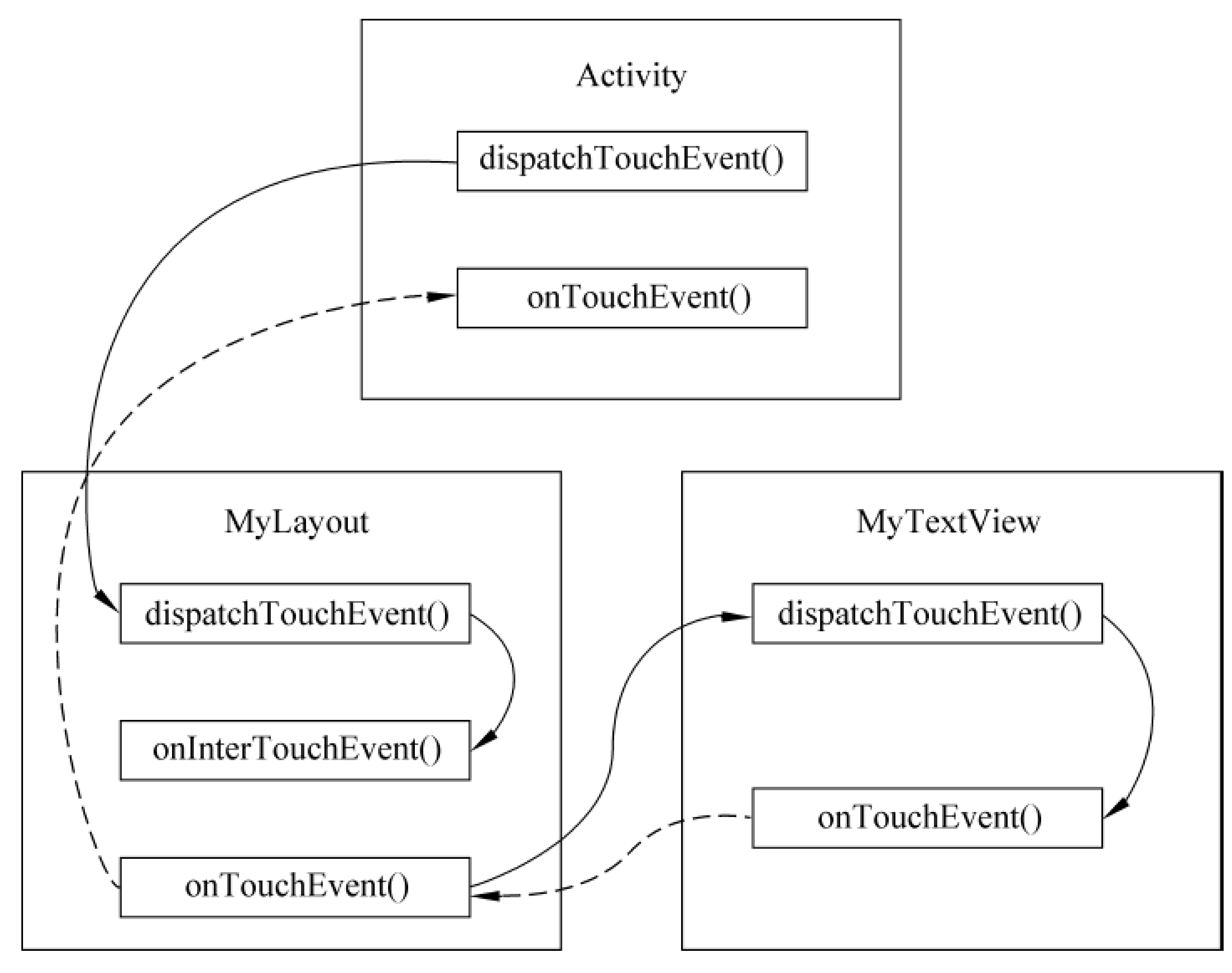


图 9.4　从内向外传递触摸事件路径

下面是这个过程的日志输出：

```
07-16 22:17:55.110: D/InterceptTouchActivity(19689): dispatchTouchEvent
action:ACTION_DOWN
07-16 22:17:55.114: D/MyLayoutView(19689): dispatchTouchEvent action:
ACTION_DOWN
07-16 22:17:55.114: D/MyLayoutView(19689): onInterceptTouchEvent action:
ACTION_DOWN
07-16 22:17:55.114: D/MyTextView(19689): dispatchTouchEvent action:ACTION
_DOWN
07-16 22:17:55.114: D/MyTextView(19689): onTouchEvent action:ACTION_DOWN
07-16 22:17:55.114: D/MyLayoutView(19689): onTouchEvent action:ACTION_DOWN
07-16 22:17:55.114: D/InterceptTouchActivity(19689): onTouchEvent action:
ACTION_DOWN
07-16 22:17:55.114: D/InterceptTouchActivity(19689): dispatchTouchEvent
action:ACTION_UP
07-16 22:17:55.114: D/InterceptTouchActivity(19689): onTouchEvent action:
ACTION_UP
```

如上面的示例一样，如果改变三种方法的返回值，会有不同的日志输出，可以尝试改变其他方法的返回值，体会一下触摸事件传递的原理。

9.2.2 同一层次之间

另一种响应触摸事件的方式，就是设置 OnTouchListener 监听器。OnTouchListener 是用来处理屏幕触摸事件的监听接口，当在 View 的范围内触摸按下、抬起或滑动等动作时都会触发该事件。需要实现该接口的方法为：

```
public boolean onTouch(View v,MotionEvent event)
```

其中，参数 v 为事件源对象；而参数 event 为手机屏幕触摸事件封装类的对象，其中封装了该事件的所有信息，例如触摸的位置、触摸的类型以及触摸的时间等，该对象会在用户触摸手机屏幕时被创建。当这个方法的返回值为 true 时，表示处理完触摸事件且不传递到其他回调方法，否则返回 false。

现在回想一下第 4 章的内容。可能有人会问，如果用户单击界面会产生触摸事件吗？如果同时为一个控件注册了 OnClickListener( )、OnLongClickListener( ) 和 OnTouchListener()，而且还覆盖了 onTouchEvent()方法，触摸事件的传递顺序又是怎样的？

实际上，当用户完成一次单击操作时，屏幕上的触摸传感器得到的按下和抬起信号，可以理解为发生了触摸事件的 ACTION_DOWN 和 ACTION_UP 操作。所以，在 Android 中的单击事件是和触摸事件相关的。如果在一个 View 中覆盖了 onClick()、onLongClick()及 onTouchEvent()方法，则 onTouchEvent()最先捕捉到 ACTION_DOWN 事件，其次才可能触发 onClick()方法或者 onLongClick()方法，以及后续的触摸

事件。

根据触摸事件处理的逻辑，下面对同时作用在控件上的监听器或 onTouchEvent()方法的三种情况进行分析：

(1) OnClickListener()、OnLongClickListener()与 onTouchEvent()同时作用在一个控件上。在实际操作中，是否执行 onClick()和 onLongClick()方法与触摸事件的 ACTION_DOWN 和 ACTION_UP 动作有关。当一个单击事件发生时，事件发生的顺序为：ACTION_DOWN --> ACTION_UP --> onClick()。

如果发生了一个长按事件，就是按下界面控件保持一段时间，然后抬起，此时事件发生的顺序为：ACTION_DOWN --> onLongClick()-->ACTION_UP。

(2) OnTouchListener()、onTouchEvent()同时作用在一个控件上。首先执行 OnTouchListener. onTouch()的方法，如果返回值为 ture，则结束触摸事件的传递；如果返回值为 false，则继续传递触摸事件到 onTouchEvent()方法。

(3) 第三种情况为 OnClickListener()、OnLongClickListener()、OnTouchListener()同时作用在一个控件上。由于执行 onLongClick()方法是由单独的线程完成的，并且在 ACTION_UP 之前，而 onClick()的发生是在 ACTION_UP 后，因此同一次用户触摸事件就有可能既发生 onLongClick()又发生 onClick()。对于这种情况，onLongClick()方法使用返回值表示是否消费了触摸事件。如果 onLongClick()方法返回结果为 true，那么 onClick 事件就没有机会被触发了。如果 onLongClick()方法返回 false 时，一次触摸事件的基本时序为：onTouch(ACTION_DOWN)-->onLongClick()-->onTouch(ACTION_UP)-->OnClick()。

可以看到，在 ACTION_UP 后仍然触发了 onClick()方法。

9.3 速率跟踪

在 Android 应用程序开发过程中，特别是游戏程序开发中，可能需要获取触摸点移动的速度。也就是当手指在触摸屏上运动时，可能希望知道其移动的速度有多快。Android 提供了一个 VelocityTracker 帮助类，用来处理触摸事件序列，跟踪手指运动的速率。

当需要跟踪速率时，首先使用 VelocityTracker 的静态方法 VelocityTracker. obtain()获得 VelocityTracker 对象，然后可以使用 VelocityTracker. addMovement(MotionEvent event)方法添加触摸事件对象。可以在接收并且处理 MotionEvent 对象的方法(例如 OnTouchListener 的 onTouch()方法或者 onTouchEvent()方法)中添加 addMovement(MotionEvent event)。

下面使用一个简单的例子，将触摸点移动的速度写入日志信息，在运行时可以从控制台看到相应的信息。具体实现见代码 9.4，其中 touch_velocity_tracker. xml 布局文件中可以只简单设置一个 TextView。

代码 9.4 使用 VelocityTracker 进行速率跟踪

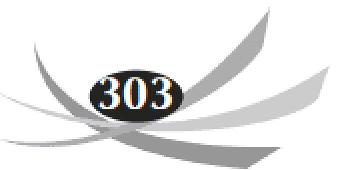
```
public class VelocityTrackerActivity extends Activity {
    private static final String TAG="VelocityTracker";

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.touch_velocity_tracker);
    }

    private VelocityTracker vTracker=null;

    public boolean onTouchEvent(MotionEvent event) {
        int action=event.getAction();
        switch(action) {
        case MotionEvent.ACTION_DOWN:
            if(vTracker==null) {
                vTracker=VelocityTracker.obtain();
            } else {
                vTracker.clear();
            }
            vTracker.addMovement(event);
            break;
        case MotionEvent.ACTION_MOVE:
            vTracker.addMovement(event);
            vTracker.computeCurrentVelocity(1000);
            Log.v(TAG, "X velocity is "+vTracker.getXVelocity()
                    +" pixels per second");
            Log.v(TAG, "Y velocity is "+vTracker.getYVelocity()
                    +" pixels per second");
            break;
        case MotionEvent.ACTION_UP:
        case MotionEvent.ACTION_CANCEL:
            vTracker.recycle();
            break;
        }
        return true;
    }
}
```

如果 VelocityTracker 中连续添加了两个 MotionEvent 对象，那么就可以计算这两个 MotionEvent 对象之间的数据变化，这些数据的变化反映了手指的运动速率。



VelocityTracker两个方法getXVelocity()和getYVelocity()分别返回手指在X和Y方向上的速率。这两个方法的返回值表示单位时间段在X和Y方向上移动的像素。这个单位时间段可以是每毫秒或秒，也可以是一个设置的任意值。

在使用这两个方法之前，需要VelocityTracker类提供的computeCurrentVelocity(int unit)方法，来设置跟踪速率时VelocityTracker所使用的时间单位。例如，如果想得到每毫秒的移动像素，给参数unit赋值为1；如果想得到每秒的移动像素，给参数unit赋值为1000。如果手指朝着X方向的右侧移动或者Y方向的底部移动，getXVelocity()和getYVelocity()方法的返回值为正；如果手指朝着X方向的左侧移动或者Y方向的顶部移动，getXVelocity()和getYVelocity()方法的返回值为负。如果结束VelocityTracker对象的使用，需要使用recycle()方法。

值得注意的是，当第一个ACTION_DOWN触摸事件被加入到VelocityTracker时，速率计算的结果虽然是零，但是还是必须首先增加这个起点，这样后续的ACTION_MOVE事件才可以计算速率。在ACTION_UP被加入后，速率的计算结果还是会变成零。因此，不要在增加了ACTION_UP触摸事件之后读取X和Y的速率。例如，当开发一个游戏程序实现用户在屏幕上扔物体时，应该在增加最后一个ACTION_MOVE事件之后，才使用速率来计算物体的轨道。

另一点就是跟踪速率很耗费资源。所以应该在使用完之后回收VelocityTracker对象，这样此对象还可以重新被使用。Android系统可以允许有多个VelocityTracker对象同时存在，但是同样会占用更多的内存。当开始跟踪一个新的触摸序列时，也可以使用clear()方法使VelocityTracker对象回到初始状态。

9.4 多点触控

在2006年的TED大会上，Jeff Han展示了一种具有多点触控技术的电脑屏幕。Android也加入了多点触控功能，目前市面上只要使用电容屏触控原理的手机均可以支持多点触控技术，可以实现图片和页面缩放、手势操作等更好的用户体验。Android的多点触控功能需要运行在Android 2.0版本以上(实际上第一个Android设备支持两个手指的多点触控)。对于这个版本，可以在屏幕上同时使用三个手指完成缩放、旋转或者任何使用多点触控想做的事情。

多点触控和单点触控的基本原理是一致的。当手指触摸屏幕时，MotionEvent对象被创建，并且被传递到前面介绍的方法中。还可以通过上面介绍的MotionEvent的getAction()、getDownTime()和getX()等方法来获取触摸事件的数据。当在屏幕上有多个触点时，MotionEvent对象必须包含所有触点的信息，但是getAction()方法得到的只是一个触点的动作值，而不是全部的触点。getDownTime()方法表示第一个手指按下的时间，如果有多个手指同时触摸屏幕，之后可能有的手指离开了屏幕，但是只要屏幕上还存在最后一个触点，这个时间值就一直保持不变。当使用getX()和getY()方法获取触摸事件发生的位置，以及使用getPressure()方法和getSize()方法获取触点压力和大小时，如果传入触点索引参数，就可以获得对应触点的信息。对于不带参数的方法调用，只能获

得第一个触点的信息。

如果希望使用多点触控,首先要知道使用 getPointerCount()方法获取当前屏幕上的触点数量。只要获得的触点数量大于 1,就需要处理触点索引和触点 ID。MotionEvent 对象中包含了当前从索引为 0 开始的触点信息,一直到 getPointerCount()方法返回的最大索引值。触点索引始终从 0 开始,如果有三个触点,则它们的索引分别为 0、1 和 2。调用类似 getX()的方法必须使用触点索引作为参数才可以获得指定的触点信息。

触点 ID 也是一个整数,可以表示哪个触点被跟踪。当第一个手指触摸屏幕时,触点 ID 也是从 0 开始的,但是随着手指从屏幕上的移走或放下,当前这个值就有可能不是从 0 开始。这是由于 Android 系统使用触点 ID 作为跟踪屏幕上不同手指的运动,触点 ID 可以固定地指定某个手指。为了说明这一点,假设由两个手指产生的一对触摸序列,序列产生的过程为:首先从手指 1 触摸开始,然后是手指 2 触摸;接着手指 1 移走,然后手指 2 移走。手指 1 触摸时,会得到触点 ID 为 0,手指 2 触摸时,会得到触点 ID 为 1,而它们的触点索引也相同。当手指 1 移走时,手指 2 的 ID 仍然为 1,而这时手指 2 的触点索引就会变成 0,这就是上面说的触点索引始终从 0 开始。这样就可以在应用程序中使用触点 ID 将触摸事件与特定的手指以及涉及到的其他手指关联起来。

在一个手势中,可以使用 getPointerId()方法获得触点 ID,用来在后续的触摸事件中跟踪手指。发生一系列的动作后,可以使用 findPointerInder()方法找到触点 ID 当前对应的触点索引,然后使用触点索引获取触摸事件的信息(见代码 9.5)。

代码 9.5 使用 getPointerId()方法

```
private int mActivePointerId;

public boolean onTouchEvent(MotionEvent event) {
    …
    //Get the pointer ID
    mActivePointerId=event.getPointerId(0);

    //… Many touch events later…

    //Use the pointer ID to find the index of the active pointer
    //and fetch its position
    int pointerIndex=event.findPointerIndex(mActivePointerId);
    //Get the pointer's current position
    float x=event.getX(pointerIndex);
    float y=event.getY(pointerIndex);
}
```

另外,可以使用 getActionMarked()方法获得触摸事件的动作。与 getAction()方法不同,这个方法是为多点触控定义的。这个方法的返回结果经过掩码处理,去掉了触点索引的信息。可以使用 getActionIndex()方法返回触摸事件的触点索引(见代码 9.6)。

代码 9.6　使用 getActionMasked()方法

```
int action=MotionEvent.getActionMasked(event);
//Get the index of the pointer associated with the action.
int index=MotionEvent.getActionIndex(event);
int xPos=-1;
int yPos=-1;

Log.d(DEBUG_TAG,"The action is "+actionToString(action));

if(event.getPointerCount()>1) {
    Log.d(DEBUG_TAG,"Multitouch event");
    //The coordinates of the current screen contact, relative to
    //the responding View or Activity.
    xPos=(int)MotionEvent.getX(event, index);
    yPos=(int)MotionEvent.getY(event, index);

} else {
    //Single touch event
    Log.d(DEBUG_TAG,"Single touch event");
    xPos=(int)MotionEvent.getX(event, index);
    yPos=(int)MotionEvent.getY(event, index);
}
…

//Given an action int, returns a string description
public static String actionToString(int action) {
    switch(action) {

        case MotionEvent.ACTION_DOWN: return "Down";
        case MotionEvent.ACTION_MOVE: return "Move";
        case MotionEvent.ACTION_POINTER_DOWN: return "Pointer Down";
        case MotionEvent.ACTION_UP: return "Up";
        case MotionEvent.ACTION_POINTER_UP: return "Pointer Up";
        case MotionEvent.ACTION_OUTSIDE: return "Outside";
        case MotionEvent.ACTION_CANCEL: return "Cancel";
    }
    return "";
}
```

9.5　手势识别

手势也是一组触摸事件的序列，由基本触摸事件的动作组成。手势可以是简单的触摸事件序列，例如单击、滑屏等，也可以是自定义更复杂的触摸事件序列。基本的手势包

括单击、长按、滑动、拖动、双击、缩放操作。每种手势都是用户的一种特定动作，触摸屏可以识别这些动作完成相应的功能。滑动就是手指在屏幕上拖动一个物体，快速地朝一个方向移动，然后抬起。在浏览图片的应用中会用到这种手势。当用户滑动触摸屏时，新的图片就会显示。

Android 提供了 GestureDetector 类检测的一些常见手势，其中的方法包括 onDown()、onLongPress()和 onFling()等。另外，使用 ScaleGestureDetector 类来实现缩放手势。

9.5.1 发现手势

当初始化 GestureDetector 对象时，需要传入一个实现了 OnGestureListener 接口的参数，当一个特定的手势被识别时，就会执行 OnGestureListener 中的各种手势的处理方法。为了使 GestureDetector 对象能够接收触摸事件，需要覆盖 View 或 Activity 的 onTouchEvent()方法，并将所有的触摸事件传递到 GestureDetector 对象中（见代码 9.7）。

代码 9.7 使用 GestureDetector 类识别手势

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG="Gestures";
    private GestureDetectorCompat mDetector;

    //Called when the activity is first created.
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //Instantiate the gesture detector with the
        //application context and an implementation of
        //GestureDetector.OnGestureListener
        mDetector=new GestureDetectorCompat(this,this);
        //Set the gesture detector as the double tap
        //listener.
        mDetector.setOnDoubleTapListener(this);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        //Be sure to call the superclass implementation
```

```
        return super.onTouchEvent(event);
    }

@Override
//单击,触摸屏按下时立刻触发
    public boolean onDown(MotionEvent event) {
        Log.d(DEBUG_TAG,"onDown: "+event.toString());
        return true;
    }

@Override
//滑动,触摸屏按下后快速移动并抬起,会先触发滚动手势,跟着触发一个滑动手势
    public boolean onFling(MotionEvent event1, MotionEvent event2,
            float velocityX, float velocityY) {
        Log.d(DEBUG_TAG, "onFling: "+event1.toString()+event2.toString());
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onLongPress: "+event.toString());
    }

@Override
//滚动,触摸屏按下后移动
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float
    distanceX, float distanceY) {
        Log.d(DEBUG_TAG, "onScroll: "+e1.toString()+e2.toString());
        return true;
    }

@Override
//短按,触摸屏按下片刻后抬起,会触发这个手势,如果迅速抬起则不会
    public void onShowPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onShowPress: "+event.toString());
    }

@Override
//抬起,手指离开触摸屏时触发(长按、滚动、滑动时,不会触发这个手势)
    public boolean onSingleTapUp(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapUp: "+event.toString());
        return true;
    }
```

```
@Override
//双击,手指在触摸屏上迅速单击第二下时触发
    public boolean onDoubleTap(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTap: "+event.toString());
        return true;
    }

@Override
//双击,按下跟抬起各触发一次
    public boolean onDoubleTapEvent(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTapEvent: "+event.toString());
        return true;
    }

@Override
//单击确认,即很快地按下并抬起,但并不连续单击第二下
    public boolean onSingleTapConfirmed(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapConfirmed: "+event.toString());
        return true;
    }
}
```

在代码 9.7 中,如果 OnGestureListener 中方法执行的结果为 true,表明已经处理触摸事件;如果结果为 false,则触摸事件继续传递,直到成功处理为止。

如果只想处理部分手势,可以继承 SimpleOnGestureListener。SimpleOnGestureListener 实现了 OnGestureListener 接口的所有方法,而且返回值都为 false。因此只需要覆盖那些关心的方法即可。无论是否使用 OnGestureListener,最好的做法就是实现 onDown() 方法,并返回 true。这是因为所有的手势都需要判断 onDown()方法的返回值,如果在 onDown()中返回 false(这是 SimpleOnGestureListener 中的默认返回结果),系统认为忽略剩余的手势动作,SimpleOnGestureListener 中的其他方法不会被调用。

9.5.2 缩放手势处理

从 Android 2.2 开始引入了 ScaleGestureDetector 类,该类可以用来识别缩放手势。缩放手势有两种操作:一种是两个手指同时触摸屏幕,向相互远离的方向移动,然后同时离开屏幕,这是放大操作;另一种是两个手指同时触摸屏幕,向相互靠近的方向移动,然后同时离开屏幕,这是缩小操作。

下面使用例子来说明如何使用缩放手势,来缩放一个图标文件。首先定义 XML 布局文件(见代码 9.8)。

代码 9.8　定义布局文件 scale_detector_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="matrix"
        android:src="@drawable/icon" />

</LinearLayout>
```

代码 9.8 布局文件中使用 ImageView 控件定义一个图片源，而且定义了一种矩阵缩放的方式。这个图片的大小是填充布局，这样图片不会被 ImageView 的边界剪切。代码 9.9 是具体实现缩放手势的 Activity 代码。

代码 9.9　实现缩放手势

```
public class ScaleDetectorActivity extends Activity {
    private static final String TAG="ScaleDetector";
    private ImageView image;
    private ScaleGestureDetector mScaleDetector;
    private float mScaleFactor=1f;
    private Matrix mMatrix=new Matrix();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.scale_detector_layout);

        image=(ImageView) findViewById(R.id.image);
        mScaleDetector=new ScaleGestureDetector(this, new ScaleListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        Log.v(TAG, "in onTouchEvent");
        //Give all events to ScaleGestureDetector
```

```
        mScaleDetector.onTouchEvent(ev);

        return true;
    }

    private class ScaleListener extends
            ScaleGestureDetector.SimpleOnScaleGestureListener {
        @Override
        public boolean onScale(ScaleGestureDetector detector) {
            mScaleFactor *=detector.getScaleFactor();

            //Make sure we don't get too small or too big
            mScaleFactor=Math.max(0.1f, Math.min(mScaleFactor, 5.0f));

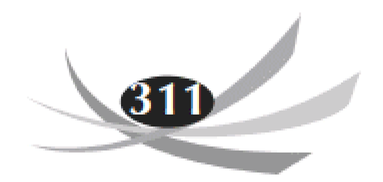
            Log.v(TAG, "in onScale, scale factor="+mScaleFactor);
            mMatrix.setScale(mScaleFactor, mScaleFactor);

            image.setImageMatrix(mMatrix);
            image.invalidate();
            return true;
        }
    }
}
```

代码 9.9 在 onCreate()方法中,获取了 ImageView 和 ScaleGestureDetector 对象,覆盖 Activity 的 onTouchEvent()方法。在这个方法中,将所有触摸事件传递给 ScaleGestureDetector 的 onTouchEvent()方法,并且结果返回 true。这样 Activity 的 onTouchEvent()方法可以不断地获得新的事件,并把事件传递给 ScaleGestureDetector,通过获得所有的触摸事件,识别出缩放手势。

例子中,在自定义 ScaleListener 的 onScale()方法中实现图片的缩放。实际上,OnScaleGestureListener 监听器中有 onScaleBegin()、onScale()和 onScaleEnd()三个回调方法,分别表示手势开始、进行和结束三个过程。

在 onScale()方法中传入 ScaleGestureDetector 对象,可以得到很多关于缩放操作的信息。mScaleFactor 是缩放因子,在 1 的上下区间(最小值为 0.1,最大值为 5.0)浮动。当两个手指靠近时,此值会小于 1;当两个手指分开时,此值会大于 1。mScaleFactor 的值从从 1 开始,随着手指的靠近或分开,图片逐渐变小或变小。如果 mScaleFactor 等于 1,图片是正常的大小。本例还为 mScaleFactor 设置了最小值和最大值,防止图片过小或过大。



9.6　拖放处理

在对触摸屏操作中，将对象拖曳穿过屏幕是常用的操作。如果 Android 系统是 Android 3.0 或以上的版本，可以使用 Android 的拖放框架，使用拖放事件监听器 View.OnDragListener 来实现。

使用 Android 的拖放框架，允许用户通过一个图形化的拖放手势，把数据从当前布局中的一个视图上转移到另一个视图上。这个框架包含了一个拖动事件类，拖动监听器和一些辅助的方法和类。

虽然这个框架主要是为了数据的移动而设计的，但是可以将这些移动的数据提供给其他的 UI 操作使用。例如，可以创建一个当用户把一个彩色图标拖到另一个彩色图标上时，将颜色混合起来的应用。

9.6.1　拖放操作

当用户执行一些被当作是开始拖动数据的信号的手势时，一个拖放动作就开始了。作为回应，应用程序告诉系统拖动动作开始了。系统回调应用程序，获取正在被拖动图形的数据，创建一个拖动图形的暗色代表图形，成为拖动阴影。当用户的手指将拖动阴影移动到当前布局上时，系统创建发送拖动事件，并传递给拖动事件监听器对象，以及与布局中 View 相联系的拖动事件回调方法。一旦用户释放这个拖动阴影，系统就结束拖动操作。

Android 应用程序在处理拖放操作时，可以通过实现 View.OnDragListener 接口，创建拖动事件监听器；然后通过 View 类提供的 setOnDragListener()方法，为 View 对象设置一个拖动事件监听器对象。每个 View 对象都可以有一个 onDragEvent() 回调方法。

应用程序通过调用 View.OnDragListener 的 startDrag()方法告诉系统开始一个拖动，也就是告诉系统可以开始发送拖动事件了。一旦应用程序调用 startDrag()方法，剩下的过程就是使用系统发送给布局中的视图对象的事件。

1. 拖放过程

拖放过程包括以下四个基本步骤或状态：

1）开始

为了响应用户开始拖动的手势，应用程序通过调用 startDrag()方法告诉系统开始一个拖动动作。startDrag()的参数提供被拖动的数据，描述被拖动数据的元数据以及一个绘制拖动阴影的回调方法。

系统首先通过回调应用程序去获得一个拖动阴影，然后将这个拖动阴影显示在设备上，接着系统发送一个操作类型为 ACTION_DRAG_STARTED 的拖动事件，给当前布局中的所有视图对象的拖动事件监听器。为了继续接收拖动事件，包括一个可能的拖动事件，拖动事件监听器必须返回 true。

如果拖动事件监听器返回值为 false，那么在当前操作中就接收不到拖动事件，直到系统发送一个操作类型为 ACTION_DRAG_ENDED 的拖动事件。通过发送 false，监听

器告诉系统它对拖动操作不感兴趣，并且不想接收被拖动的数据。

2）继续

用户继续拖动。当拖动阴影和视图对象的边界框相交时，系统会发送一个或多个拖动事件给视图对象的拖动事件监听器，当然该事件监听器已经注册。作为回应，监听器可以选择改变响应拖动事件的视图对象的外观。例如，如果事件表明阴影已经进入了视图的边界框（操作类型为 ACTION_DRAG_ENDED），那么监听器就可以高亮视图以作出回应。

3）释放

用户在可以接收数据的视图的边界框内释放拖动阴影。系统发送一个操作的类型为 ACTION_DROP 拖动事件给视图对象的监听器。这个拖动事件包括调用 startDrag()方法传给系统的数据。如果接收释放动作的代码执行成功，那么这个监听器会被期望返回 true 给系统。

值得注意的是，只有接收拖动事件的视图已经注册了监听器，用户在这个视图的边界框内释放这个拖动阴影时，ACTION_DROP 事件才会发生。如果用户在其他情况下释放这个拖动阴影，ACTION_DROP 的拖动事件就不会被发送。

4）终止

在用户释放拖动阴影并且系统发送出一个操作类型为 ACTION_DROP 的拖动事件之后，系统发送出一个操作类型为 ACTION_DRAG_ENDED 的事件来表明这个拖动操作已经结束了。不管用户在哪里释放这个拖动阴影，这个步骤都会发生。这个事件会发送给每一个被注册为接收拖动事件的监听器，无论其是否接收了拖动视图。

2. 拖放事件

用户界面的视图对象通过实现 View.OnDragListener 接口的拖动事件监听器，或通过它自身的 onDragEvent(DragEvent)回调方法来接收拖动事件。当系统回调这个方法或监听器时，会传递给它们一个拖动事件的对象。在 Android 系统中，使用 DragEvent 类来描述拖动事件。

在大多数情况下，可能会想要使用监听器。因为实现一个监听器类，可以在几个不同的视图对象中使用它。当然，也可以将这个监听器类作为一个匿名内部类去实现。实现后的监听器需要在视图对象上注册，该对象才能够接收监听到拖动事件。注册监听器可以通过调用视图的 setOnDragListener()方法来实现。

视图对象可以同时有一个监听器和一个回调方法。如果在这种情况下，系统会首先调用监听器。除非监听器返回的是 false，否则系统不会去调用回调方法。onDragEvent(DragEvent)方法和 View.OnDragListener 的结合跟触屏事件的 onTouchEvent()与 View.OnTouchListener 的结合是相似的。

当拖动事件发生时，系统创建一个 DragEvent 对象，并传递给相应的回调方法或监听器。这个对象包括拖放事件中正在发生事件的类型以及其他依赖这个事件类型的数据。监听器调用 getAction()方法就可以获得这个事件类型（见表 9.2）。

表 9.2　DragEvent 的事件类型

getAction()的值	意　　义
ACTION_DRAG_STARTED	在应用程序调用 startDrag()并获得一个拖动阴影之后，视图对象的拖动事件监听器就会接收到这个事件类型的事件
ACTION_DRAG_ENTERED	当拖动阴影刚刚进入视图的边界框范围时，视图的拖动事件监听器就会接收到这个动作类型的事件。这是当拖动阴影进入视图的边界框范围时监听器所接收到的第一个事件操作类型。如果监听器还继续为拖动阴影进入视图边界框范围这个动作接收拖动事件，那么必须返回 true 给系统
ACTION_DRAG_LOCATION	当拖动阴影还在视图的边界框范围中，视图的拖动事件监听器就会在接收到 ACTION_DRAG_ENTERED 事件之后接收到这个操作类型的事件
ACTION_DRAG_EXITED	当视图的拖动事件监听器接收到 ACTION_DRAG_ENTERED 事件，并且至少接收到一个 ACTION_DRAG_LOCATION 事件，那么在用户把拖动阴影移除视图的边界框范围之后，该监听器就会再接收到这个操作类型的事件
ACTION_DROP	当用户在视图对象上释放拖动阴影时，该视图对象的拖动事件监听器就会接收到这个类型的拖动事件。这个操作类型只会发送给在回应 ACTION_DRAG_STARTED 类型的拖动事件中返回 true 的那个视图对象的监听器。如果用户释放拖动阴影的那个视图没有注册监听器，或者用户在当前布局之外的任何对象上释放了拖动阴影，那么这个操作类型就不会被发送。如果释放动作顺利，监听器应该返回 true，否则应该返回 false
ACTION_DRAG_ENDED	当系统结束拖动动作时，视图对象的拖动事件监听器就会接收到这个类型的拖动事件。这种操作类型前面不一定有一个 ACTION_DROP 事件。如果系统发送一个 ACTION_DROP，并接收到一个 ACTION_DRAG_ENDED 操作类型，并不意味着拖动事件的成功。监听器必须调用 getResult()方法来获取在回应 ACTION_DROP 事件中返回的结果。如果 ACTION_DROP 事件没有被发送，那么 getResult()就返回 false

3. 拖放阴影

在拖动过程中，系统会显示一张用户拖动的图片。对数据移动而言，这张图片代表着那些正在被移动的数据。对其他操作而言，这张图片代表着拖动操作的某些环节。这张图片就被叫做一个拖动阴影。

拖动阴影可以通过在 View. DragShadowBuilder 中定义的方法创建，并调用应用程序定义的 View. DragShadowBuilder 里面的回调方法去获取一个拖动阴影。

View. DragShadowBuilder 类有两个构造方法：

- View. DragShadowBuilder(View)。

此构造方法接收的应用程序中的任意一个视图对象，并将此视图对象存储在 View.

DragShadowBuilder 对象中。因此在回调过程中，可以直接把它作为拖动阴影。构造方法不必和用户选择开始一个拖动的视图对象(如果有的话)相关联。

如果使用这个构造方法，不必去继承 View. DragShadowBuilder 类或覆盖它的方法。默认情况下，会得到一个与作为参数传递的那个视图有相同外表的拖动阴影，并且该拖动阴影会居中位于用户接触的屏幕上。

- View. DragShadowBuilder()。

如果使用该构造方法，在 View. DragShadowBuilder 对象中没有一个视图对象是有效的（这个字段被设置为 null）。如果使用该构造方法，不必继承 View. DragShadowBuilder 类或覆盖它的方法，可以得到一个不可见的拖动阴影。系统不会给出一个错误。

View. DragShadowBuilder 类有两个方法：

- onProvideShadowMetrics()。

系统在调用了 android. view. View. DragShadowBuilder 的 startDrag()方法之后，立刻回调用该方法。用该方法给系统发送拖动阴影的规模和接触点。该方法有两个参数，其中 dimensions 表示一个 Point 对象，指拖动阴影的宽为 x，高为 y；另一个参数 touch_point 表示一个 Point 对象，指拖动过程中，在用户手指之下的拖动阴影的位置，X 轴坐标为 x，Y 轴坐标为 y。

- onDrawShadow()。

在调用了 onProvideShadowMetrics()方法之后，系统立刻调用 onDrawShadow()这个方法来获取拖动阴影。这个方法只有一个参数，一个 Canvas 对象，该对象是系统利用提供给 onProvideShadowMetrics()方法里面的参数构造出来的。利用它可以在提供给的 Canvas 对象中绘制拖动阴影。

9.6.2 设计拖动操作

本节说明如何开始一个拖动，如何在拖动过程中回应事件，如何回应一个拖动事件以及如何结束一个拖放操作。

1. 开始拖动

用户用一个拖动的手势开始一个拖动，通常是一个在视图对象上的长按动作。作为回应，需要做下面两件事情。

(1) 为要移动的数据创建一个 ClipData 和 ClipData. Item 对象。

当用户在一个视图上有一个长按动作时，需要创建一个 ClipData 和 ClipData. Item 对象，用于存储在 ClipDescription 对象中的元数据。因为一个拖放动作不能表示数据的移动，可使用 null 来代替一个实际的数据。

例如，下面一个例子说明了当在 ImageView 上有一个长按动作事件时，如何创建一个 ClipData 对象，来包含这个 ImageView 的标志或标签(见代码 9.10)。以下就是这些片段，第二个片段说明了如何重写 View. DragShadowBuilder 类中的方法。



代码 9.10　创建 ClipData 对象

```
//Create a string for the ImageView label
private static final String IMAGEVIEW_TAG="icon bitmap"

//Creates a new ImageView
ImageView imageView=new ImageView(this);

//Sets the bitmap for the ImageView from an icon bit map (defined elsewhere)
imageView.setImageBitmap(mIconBitmap);

//Sets the tag
imageView.setTag(IMAGEVIEW_TAG);

    ...

//Sets a long click listener for the ImageView using an anonymous listener
object that
//implements the OnLongClickListener interface
imageView.setOnLongClickListener(new View.OnLongClickListener() {

    //Defines the one method for the interface, which is called when the View
is long-clicked
    public boolean onLongClick(View v) {

    ClipData.Item item=new ClipData.Item(v.getTag());

    ClipData dragData=new ClipData(v.getTag(),ClipData.MIMETYPE_TEXT_
    PLAIN,item);

    //Instantiates the drag shadow builder.
    View.DrawShadowBuilder myShadow=new MyDragShadowBuilder(imageView);

    //Starts the drag

            v.startDrag(dragData,    //the data to be dragged
                        myShadow,    //the drag shadow builder
                        null,        //no need to use local data
                        0            //flags (not currentlyused, set to 0)
            );

    }
}
```

（2）创建拖动阴影。

创建一个 View. DragShadowBuilder 的子类 MyDragShadowBuilder，覆盖 onDrawShadow()方法和 onProvideShadowMetrics()方法，为拖动一个 TextView 创建一个小的灰色矩形框拖动阴影(见代码 9.11)。

代码 9.11　覆盖 View. DragShadowBuilder 的方法

```
private static class MyDragShadowBuilder extends View.DragShadowBuilder {

  //The drag shadow image, defined as a drawable thing
  private static Drawable shadow;

  //Defines the constructor for myDragShadowBuilder
  public MyDragShadowBuilder(View v) {
    super(v);
    //Creates a draggable image that will fill the Canvas provided by the
    system.
    shadow=new ColorDrawable(Color.LTGRAY);
  }

   @Override
   public void onProvideShadowMetrics (Point size, Point touch)
     private int width, height;

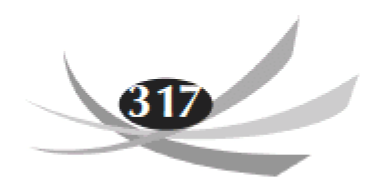
     width=getView().getWidth() / 2;
     height=getView().getHeight() / 2;

     //The drag shadow is a ColorDrawable. This sets its dimensions to be
     //the same as the Canvas that the system will provide. As a result, the
     //drag shadow will fill the Canvas.
     shadow.setBounds(0, 0, width, height);

     size.set(width, height);

     //Sets the touch point's position to be in the middle of the drag shadow
     touch.set(width / 2, height / 2);
   }

   @Override
   public void onDrawShadow(Canvas canvas) {
     //Draws the ColorDrawable in the Canvas passed in from the system.
     shadow.draw(canvas);
   }
}
```



其中 onProvideShadowMetrics()方法中的代码实现了回调方法，把拖动阴影的位置和触摸点传递给系统；onDrawShadow()方法的代码实现了回调方法，基于系统，根据 onProvideShadowMetrics()方法中传递的位置，在所构建的画布上画拖动阴影。

继承 View. DragShadowBuilder 和覆盖其方法的过程并不是必需的，构造方法 View. DragShadowBuilder(View)会创建一个默认的拖动阴影，这个拖动阴影与传递给它的 View 参数一样大，并且位于以接触点为中心的位置。

2. 响应拖动开始事件

在拖动过程中，系统将拖动事件传递给当前布局中的视图对象的拖动事件监听器。监听器应该调用 getAction()方法获取操作类型。在一个拖动开始时，该方法返回 ACTION_DRAG_STARTED。

当 ACTION_DRAG_STARTED 事件发生时，监听器需要进行下面的处理：

(1) 调用 getClipDescription()方法获取 ClipDescription。

使用在 ClipDescription 中的 MIME 类型的方法查看监听器是否接收被拖动的数据。如果拖放操作没有数据移动，这个步骤就不是必需的。

(2) 如果监听器可以接收一个拖动事件，它必须返回 true。

所谓监听器可以接收一个拖动事件，即这个监听器可以对拖动事件进行处理。设置返回值为 true，告诉系统继续发送拖动事件给监听器。如果监听器不接收一个拖动，就会返回 false，系统就会停止发送拖动事件，直到 ACTION_DRAG_ENDED 事件发生。

对于 ACTION_DRAG_STARTED 事件，有一些拖动事件的方法是无效的，例如 getClipData()、getX()、getY()和 getResult()。

3. 在拖动过程中处理事件

在拖动过程中，当监听器对 ACTION_DRAG_STARTED 拖动事件的返回值为 true 时，监听器继续接收后续的拖动事件。监听器在拖动过程中接收到的拖动事件类型取决于拖放阴影的位置以及监听器视图的可见性。

在拖动过程中，getAction()返回的事件类型包括三个：

1) CTION_DRAG_ENTERED

当接触点，即屏幕上位于用户手指下的点，进入监听器的视图的边界框范围内时，监听器会接收到该事件。

2) ACTION_DRAG_LOCATION

一旦监听器接收到 ACTION_DRAG_LOCATION 事件，在它接收到 ACTION_DRAG_EXITED 事件之前，接触点每移动一次，它都会接收到一个新的 ACTION_DRAG_LOCATION 事件。方法 getX()和 getY()会返回接触点的 X 轴和 Y 轴的坐标。

3) ACTION_DRAG_EXITED

在拖动阴影不再位于监听器视图的边界框范围之内时，这个事件会被发送给以前接收到 ACTION_DRAG_ENTERED 事件的监听器。

当这些事件发生时，监听器可以不对任意一个事件做出反应。如果监听器返回一个值给系统，它也会被忽略掉。

4. 响应释放动作

当用户在某个视图上释放拖动阴影时，该视图会预先报告是否可以接收被拖动的内容，系统会将拖动事件分发给具有 ACTION_DROP 操作类型的视图。监听器在事件处理时，需要做两件事。

一是调用 getClipData()方法获取最初在 startDrag()方法中应用的 ClipData 对象，并存储。如果拖放操作没有数据的移动，就不必进行该操作。

二是，如果释放动作已顺利完成，监听器应返回 true；如果没有完成，则返回 false。这个被返回的值成为 ACTION_DRAG_ENDED 事件中 getResult()方法的返回值。

需要注意的是，如果系统没有发送出 ACTION_DROP 事件，那么 ACTION_DRAG_ENDED 事件中 getResult()方法的返回值就为 false。

对于 ACTION_DROP 事件来说，在释放动作的瞬间，getX()和 getY()方法使用接收释放动作的视图上的坐标系统，返回拖动点的 X 轴和 Y 轴的坐标。

系统允许用户在监听器不接收拖动事件的视图上释放拖动阴影，允许用户在应用程序 UI 的空区域或者应用程序之外的区域释放拖动阴影。

5. 回应一个拖动的结束

用户释放了拖动阴影后，系统会立即给应用程序中所有的拖动事件监听器发送 ACTION_DRAG_ENDED 类型的拖动事件，表明拖动动作结束。

当 ACTION_DRAG_ENDED 事件发生时，监听器需要进行如下处理：如果监听器在操作期间改变了 View 对象的外观，应该把 View 对象重置为默认的外观，监听器向系统返回 true。监听器也可以调用 getResult()方法来查找更多的相关操作。如果在响应 ACTION_DROP 类型的事件中监听器返回了 true，那么 getResult()方法也会返回 true。在其他的情况中，getResult()方法会返回 false，包括系统没有发送 ACTION_DROP 事件的情况。

9.6.3 实现拖动操作

9.6.2 节介绍了如何在拖放操作的各个阶段对事件进行处理，下面使用一个简单的例子来说明如何实现拖放操作。

实现拖放操作的步骤共有六步，包括创建应用程序可以根据实际的情况删减步骤。

1. 定义 XML 绘制图片

在 res 的 drawable 目录下，创建 shape.xml 文件，作为正常的背景设置(见代码 9.12)。

代码 9.12 shape.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">

    <stroke
        android:width="2dp"
        android:color="#FFFFFFFF" />
```

```
    <gradient
        android:angle="225"
        android:endColor="#DD2ECCFA"
        android:startColor="#DD000000" />
    <corners
        android:bottomLeftRadius="7dp"
        android:bottomRightRadius="7dp"
        android:topLeftRadius="7dp"
        android:topRightRadius="7dp" />

</shape>
```

XML 文件中的＜shape＞元素用于定义形状。其子元素＜stroke＞定义＜shape＞中线的属性。子元素＜gradient＞定义该形状里面为渐变色填充，startColor 为起始颜色，endColor 为结束颜色，angle 表示方向角度。子元素＜corners＞为 shape 创建圆角。只有当形状为矩形时才能使用。

另外还有子元素＜padding＞用于填充应用的视图对象（而不是形状），子元素＜size＞用于定义 shape 的尺寸，子元素＜solid＞用于定义填充颜色。各子元素分别具有自己的属性，可以根据设计来设定。

在 res 的 drawable 目录下，创建 shape_droptarget.xml 文件，作为当被拖动对象进入到目标对象的范围内后目标对象的背景（见代码 9.13）。

代码 9.13　shape_droptarget.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">

    <stroke
        android:width="2dp"
        android:color="#FFFF0000" />

    <gradient
        android:angle="225"
        android:endColor="#DD2ECCFA"
        android:startColor="#DD000000" />

    <corners
        android:bottomLeftRadius="7dp"
        android:bottomRightRadius="7dp"
        android:topLeftRadius="7dp"
        android:topRightRadius="7dp" />

</shape>
```

2. 定义布局等资源文件

定义 Activity 显示的用户界面 main.xml 布局文件，界面使用网格布局，每个网格中使用前面定义的 shape 形状作为背景，放置一个图片（见代码 9.14）。

代码 9.14　布局文件

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2"
    android:columnWidth="300dp"
    android:orientation="vertical"
    android:rowCount="2"
    android:stretchMode="columnWidth">

    <LinearLayout
        android:id="@+id/topleft"
        android:layout_width="160dp"
        android:layout_height="200dp"
        android:layout_column="0"
        android:layout_row="0"
        android:background="@drawable/shape">

        <ImageView
            android:id="@+id/myimage1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="0"
            android:layout_row="0"
            android:src="@drawable/ic_launcher" />
    </LinearLayout>

    <LinearLayout
        android:id="@+id/topright"
        android:layout_width="160dp"
        android:layout_height="200dp"
        android:layout_column="1"
        android:layout_row="0"
        android:background="@drawable/shape">

        <ImageView
            android:id="@+id/myimage2"
            android:layout_width="wrap_content"
```




```
            android:layout_height="wrap_content"
            android:layout_column="0"
            android:layout_row="0"
            android:src="@drawable/ic_launcher" />
    </LinearLayout>

    <LinearLayout
        android:id="@+id/bottomleft"
        android:layout_width="160dp"
        android:layout_height="200dp"
        android:layout_column="0"
        android:layout_row="1"
        android:background="@drawable/shape">

        <ImageView
            android:id="@+id/myimage3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/ic_launcher" />
    </LinearLayout>

    <LinearLayout
        android:id="@+id/bottomright"
        android:layout_width="160dp"
        android:layout_height="200dp"
        android:layout_column="1"
        android:layout_row="1"
        android:background="@drawable/shape">

        <ImageView
            android:id="@+id/myimage4"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="0"
            android:layout_row="0"
            android:src="@drawable/ic_launcher" />
    </LinearLayout>

</GridLayout>
```

3. 创建或打开 Activity，获取定义的视图对象

创建 DragActivity 作为主界面，导入布局文件 main.xml（见代码 9.15）。

代码 9.15　主 Activity

```
public class DragActivity extends Activity {

/** Called when the activity is first created. */

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

4. 定义或实现 TouchListener

在 DragActivity 中自定义一个 OnTouchListener 的监听器 MyTouchListener，当接收到 ACTION_DOWN 事件时，创建一个 ClipData 对象，使用 View. DragShadowBuilder 给当前所触摸的对象创建一个拖动阴影，设置开始拖动操作(见代码 9.16)。

代码 9.16　自定义 OnTouchListener

```
private final class MyTouchListener implements OnTouchListener {
    public boolean onTouch(View view, MotionEvent motionEvent) {
      if(motionEvent.getAction()==MotionEvent.ACTION_DOWN) {
        ClipData data=ClipData.newPlainText("", "");
        DragShadowBuilder shadowBuilder=new View.DragShadowBuilder(view);
        view.startDrag(data, shadowBuilder, view, 0);
        view.setVisibility(View.INVISIBLE);
        return true;
      } else {
        return false;
      }
    }
}
```

在这里也可以使用自定义 OnLongClickListener 来设置开始拖动操作，在其 onLongClick()方法中创建 ClipDate 对象、创建拖动阴影，以及进行其他的相关初始设置。

5. 定义或实现 DragListener

在 DragActivity 中自定义一个 OnDragListener 的监听器 MyDragListener，定义在拖动操作时，各事件发生时视图对象的工作(见代码 9.17)。当 ACTION_DRAG_STARTED 发生时，因为前面在 onTouch()代码中已经做了拖动开始操作时的一些处理，这里就不再做任何处理；当被拖动的对象进入本视图对象的 ACTION_DRAG_ENTERED 事件发生时，将背景改变为 shape_droptarget. xml 所定义的图片；当拖动释放的 ACTION_DROP 事件发生时，将原来的拖动对象删除，加入到所拖动到的位置；当

拖动操作结束的 ACTION_DRAG_ENDED 和 ACTION_DRAG_EXITED 事件发生时，将背景改回 shape.xml 定义的正常背景。

代码 9.17　自定义 OnDragListener

```
class MyDragListener implements OnDragListener {
    Drawable enterShape=getResources().getDrawable(R.drawable.shape_
    droptarget);
    Drawable normalShape=getResources().getDrawable(R.drawable.shape);

    @Override
    public boolean onDrag(View v, DragEvent event) {
      int action=event.getAction();
      switch(event.getAction()) {
      case DragEvent.ACTION_DRAG_STARTED:
        //Do nothing
        break;
      case DragEvent.ACTION_DRAG_ENTERED:
        v.setBackgroundDrawable(enterShape);
        break;
      case DragEvent.ACTION_DRAG_EXITED:
        v.setBackgroundDrawable(normalShape);
        break;
      case DragEvent.ACTION_DROP:
        //Dropped, reassign View to ViewGroup
        View view=(View) event.getLocalState();
        ViewGroup owner=(ViewGroup) view.getParent();
        owner.removeView(view);
        LinearLayout container=(LinearLayout) v;
        container.addView(view);
        view.setVisibility(View.VISIBLE);
        break;
      case DragEvent.ACTION_DRAG_ENDED:
        v.setBackgroundDrawable(normalShape);
      default:
        break;
      }
      return true;
    }
}
```

6. 将监听器注册到视图对象

在 DragActivity 中，使用 findViewById()方法获取布局文件中定义的图片对象，并通过视图对象的 setOnTouchListener()和 setOnDragListener()方法，将前面所定义的监听器对象注册到视图对象上，完成整个程序的编写(见代码 9.18)。

代码 9.18　DragActivity. java

```
public class DragActivity extends Activity {

/** Called when the activity is first created. */

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    findViewById(R.id.myimage1).setOnTouchListener(new MyTouchListener());
    findViewById(R.id.myimage2).setOnTouchListener(new MyTouchListener());
    findViewById(R.id.myimage3).setOnTouchListener(new MyTouchListener());
    findViewById(R.id.myimage4).setOnTouchListener(new MyTouchListener());
    findViewById(R.id.topleft).setOnDragListener(new MyDragListener());
    findViewById(R.id.topright).setOnDragListener(new MyDragListener());
    findViewById(R.id.bottomleft).setOnDragListener(new MyDragListener());
    findViewById(R.id.bottomright).setOnDragListener(new MyDragListener());

  }

  private final class MyTouchListener implements OnTouchListener {
    public boolean onTouch(View view, MotionEvent motionEvent) {
      if(motionEvent.getAction()==MotionEvent.ACTION_DOWN) {
        ClipData data=ClipData.newPlainText("", "");
        DragShadowBuilder shadowBuilder=new View.DragShadowBuilder(view);
        view.startDrag(data, shadowBuilder, view, 0);
        view.setVisibility(View.INVISIBLE);
        return true;
      } else {
        return false;
      }
    }
  }

  class MyDragListener implements OnDragListener {
     Drawable enterShape = getResources().getDrawable(R.drawable.shape_
     droptarget);
    Drawable normalShape=getResources().getDrawable(R.drawable.shape);

    @Override
    public boolean onDrag(View v, DragEvent event) {
      int action=event.getAction();
```

```
        switch(event.getAction()) {
        case DragEvent.ACTION_DRAG_STARTED:
          //Do nothing
          break;
        case DragEvent.ACTION_DRAG_ENTERED:
          v.setBackgroundDrawable(enterShape);
          break;
        case DragEvent.ACTION_DRAG_EXITED:
          v.setBackgroundDrawable(normalShape);
          break;
        case DragEvent.ACTION_DROP:
          //Dropped, reassign View to ViewGroup
          View view= (View) event.getLocalState();
          ViewGroup owner= (ViewGroup) view.getParent();
          owner.removeView(view);
          LinearLayout container= (LinearLayout) v;
          container.addView(view);
          view.setVisibility(View.VISIBLE);
          break;
        case DragEvent.ACTION_DRAG_ENDED:
          v.setBackgroundDrawable(normalShape);
        default:
          break;
        }
        return true;
      }
    }
  }
```

9.7　本 章 小 结

本章主要介绍了 Android 系统对于触摸屏的操作和处理。

Android 系统中把用户对触摸屏的操作定义成不同的手势。手势是指系统可识别的,用户在对屏幕显示对象操作时,手指在触摸屏上抬起、按下或是移动的方式。在 Android 系统中支持的核心手势包括触摸(Touch)、长按(Long Press)、滑动(Swipe)、拖曳(Drag)、双击(Double Touch)、放大(Pinch Open)和缩小(Pinch Close)。

在 Android 系统中,触摸事件由 MotionEvent 类来描述。产生一个触摸事件,系统就会创建一个 MotionEvent 对象。从用户手指触摸设备屏幕开始,到手指离开设备屏幕结束,Android 系统会产生一系列与手指运动相关的触摸事件,每个触摸事件都记录手指运动的信息,称这些触摸事件为一个事件序列。每一个手势都是一个事件序列。

触摸事件的处理遵循触摸事件的传递和消费机制。触摸事件在用户界面的 View 和 ViewGroup 的相邻层次之间传递，传递方向先从外向内，然后从内向外。从外向内传递就是从最外层的根元素依次递归向其包含的子元素传递，一直到最内层子元素，或中间某个元素消费了触摸事件，结束了传递；从内向外就是从最内层子元素依次递归向外层传递，直到根元素或中间某个元素消费了触摸事件，结束了传递。

对于触摸事件的处理包括速率跟踪、多点触控、手势识别和拖放处理。事件处理遵循触摸事件的传递和消费机制，具体实现可以采用事件监听的方式，也可以采用回调方法的方式处理。

第 10 章 定位服务与 Google 地图

10.1 定位服务

Android 通过 android. location 包中的类为应用程序提供定位服务。定位框架中的核心组件就是 LocationManager 系统服务，其提供了支撑底层设备的定位 API。与其他系统服务一样，并不是直接实例化一个 LocationManager 对象，而是通过调用 Context 类的 getSystemService(Context. LOCATION_SERVICE)方法来获得一个 LocationManager 对象。这个方法会返回一个新的 LocationManager 对象。

应用程序获取一个 LocationManager 对象后，就可以进行定位服务的各种操作，例如：

- 查询所有定位提供者列表，获得最新的用户位置信息。
- 周期性地注册、更新或注销用户当前位置。
- 如果设备进入到一个给定经度和纬度邻近范围时(指定一个半径)时，注册或注销一个需要启动的 Intent。

使用 Google 地图的 Android API，可以使用 Google 地图数据，将地图功能集成到应用中。API 自动处理对 Google 地图服务器的访问、数据下载、地图显示和在地图上触控手势。

10.1.1 获取位置信息

Android 设备获取位置可以使用 GPS 和 Android 网络位置提供器(Android Network Location Provider，NLP)。尽管 GPS 定位更精确，但它只能在户外使用、耗电严重，并且其返回用户位置的速度远不能满足用户需求。网络位置提供器通过基站和 Wi-Fi 信号来获取位置信息，并且室内外均可使用，其速度更快、耗电更少。为了获取用户位置信息，可以同时使用 GPS 和 Android 网络位置提供器，也可以二者任选其一。

1. 获取位置信息时要解决的问题

那么哪些因素决定了定位呢？获得用户信息是一个复杂的过程，有时候会发现获取的位置信息是错误的或者精度不高，原因有以下几种：

1) 多种位置源

GPS、Cell-ID 和 Wi-Fi 都可以提供用户位置信息，每种源的精度是不同的，但是决定使用哪个源，需要权衡精度、速度和电池的容量。

2）用户的移动

当用户移动时，因为用户位置的改变，必须经常定期获取用户位置，所以当用户在移动时，如果获取信息的频次越高，则用户位置信息越精确，但是高频次也会影响设备的运行效率和电量。

3）变化的精度

从每个位置源获得的位置估算在精度方面也是不一致的。例如，从一个位置上，10s 前获得的位置或许比从相同的或者不同的源上获取的最新位置精度更高。

2. 实现定位功能的重要类

在使用位置服务开发应用时，上面这些因素都需要考虑。但首先需要知道怎样获取位置信息。以下是 android.location 包中几个关于定位功能的比较重要的类：

1）LocationManager

该类提供访问系统定位服务。定位服务可以为应用程序提供周期性的设备的地理位置更新信息，或当设备进入某个地理范围时，发送应用程序说明的 Intent。

2）LocationProvider

它是一个抽象类，是不同定位提供者的父类，提供当前位置信息，并存储在 Location 类中。Android 设备有一些可用的 LocationProvider，表 10.1 列出了主要的 LocationProvider。

表 10.1　LocationProvider

LocationProvider	描　　述
network	使用移动网络或 Wi-Fi 来确定最佳位置，在室内精度比 GPS 高
gps	使用 GPS 接收器来确定最佳位置，通常比网络精度更高
passive	允许参与其他组件位置更新以节省能源

3）LocationListener

该类提供定位信息发生改变时的回调功能。必须事先在定位管理器中注册监听器对象。

4）Criteria

该类使得应用能够通过在 LocationProvider 中设置的属性来选择合适的定位提供者。

3. 请求位置更新信息

在 Android 中，可以通过回调的方法得到用户位置。使用 LocationManager 类，向其 requestLocationUpdates()方法传入一个 LocationListener 对象，就可以获得位置更新。在 LocationListener 中，必须要实现响应的几个回调方法，以便当用户位置信息和服务状态变化时 LocationManager 调用。

代码 10.1 使用一个简单的例子，说明了如何定义一个 LocationListener，并且请求位置更新。



代码 10.1　位置更新信息获取

```
//Acquire a reference to the system Location Manager
LocationManager locationManager= (LocationManager) this.getSystemService
(Context.LOCATION_SERVICE);

//Define a listener that responds to location updates
LocationListener locationListener=new LocationListener() {
  public void onLocationChanged(Location location) {
   //Called when a new location is found by the network location provider.
   makeUseOfNewLocation(location);
  }

  public void onStatusChanged(String provider, int status, Bundle extras) {}

  public void onProviderEnabled(String provider) {}

  public void onProviderDisabled(String provider) {}
};

//Register the listener with the Location Manager to receive location updates
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0,
0, locationListener);
```

requestLocationUpdates()方法的第一个参数是位置服务的类型，也就是程序通过什么来获取用户的位置信息；第二个参数是两次位置提醒之间的最小时间间隔；第三个参数是两次位置提醒之间最小距离间隔(第二、三两个参数都为 0 表示尽可能频繁地请求位置信息)；第四个参数为 LocationListener。例如每隔 30s 收集一次 GPS 信息，可以用下面的代码实现：

```
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,30 *
1000, 0, myListenGPS);
```

代码 10.1 中，选择的位置服务类型为 NETWORK_PROVIDER，而 Android 系统提供两种位置服务类型，其中包括：LocationManager.GPS_PROVIDER 和 LocationManager.NETWORK_PROVIDER。

应用程序如果要使用这两种方式的定位服务，需要通过系统设置，见图 10.1。

这两种方式的区别是什么呢？GPS_PROVIDER 提供精确的 GPS 定位，但在室内几乎无法定位而导致无法收集信息，即有定位盲区。GPS 定位的基本原理是测量出已知位置的卫星到用户接收机之间的距离，然后综合多颗卫星的数据就可知道接收机的具体位置。要达到这一目的，卫星的位置可以根据星载时钟所记录的时间在卫星星历中查出，所以使用必须在户外。而 NETWORK_PROVIDER 为网络定位，其偏差较大，但无定位盲

区，只要有网络一般都可以收集到。网络定位简单来说就是当前接入 Wi-Fi 就使用 Wi-Fi 定位，当前接入 2G 或 3G 网就是基站定位，实际上基站和 Wi-Fi 有单独的定位方式，只不过系统都封装到了 NETWORK_PROVIDER 方法中。



图 10.1　设置定位方式

除了 requestLocationUpdates()方法，LocationManager 类还提供了 getLastKnownLocation()方法，来获取上一次获取到的位置信息，而并非当前的 GPS 位置信息。

4. 用户权限设置

为了从 NETWORK_PROVIDER 或 GPS_PROVIDER 获取位置更新，必须在应用程序的 Manifest 文件中声明用户访问的 ACCESS_COARSE_LOCATION 或 ACCESS_FINE_LOCATION 权限(见代码 10.2)。

代码 10.2　定位服务用户权限设置

```
<manifest …>
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
…
</manifest>
```

如果在应用程序中同时使用 NETWORK_PROVIDER 和 GPS_PROVIDER，就只需声明 ACCESS_FINE_LOCATION 权限。ACCESS_COARSE_LOCATION 只包含 NETWORK_PROVIDER 的权限。

10.1.2　定位最佳策略

基于位置的应用可谓数不胜数，但是由于很难提供最佳精度、用户位置的移动、多种方法获取用户位置和尽可能减少耗电量等原因，使得获取用户位置变得较为复杂。要既减少电池的耗电量，同时又获取极佳用户位置，必须定义一个长效模型来解决多种难题，说明应用如何获取用户的位置。当启动或停止监听位置更新，或使用缓存位置数据时，此模型会被使用。下面是获取用户位置的典型流程：

(1) 启动应用。

(2) 一段时间后，开始监听定位提供者获取位置信息。

(3) 通过去除不够准确的位置更新来保持以最佳状态去获取位置信息。

(4) 停止监听获取位置信息。

(5) 采用最新最好的位置。

图 10.2 通过使用时间线展示了获取用户位置更新的流程时间线。这个时间线体现

了应用监听用户位置更新的各个时间段和各个时间段发生的事件。

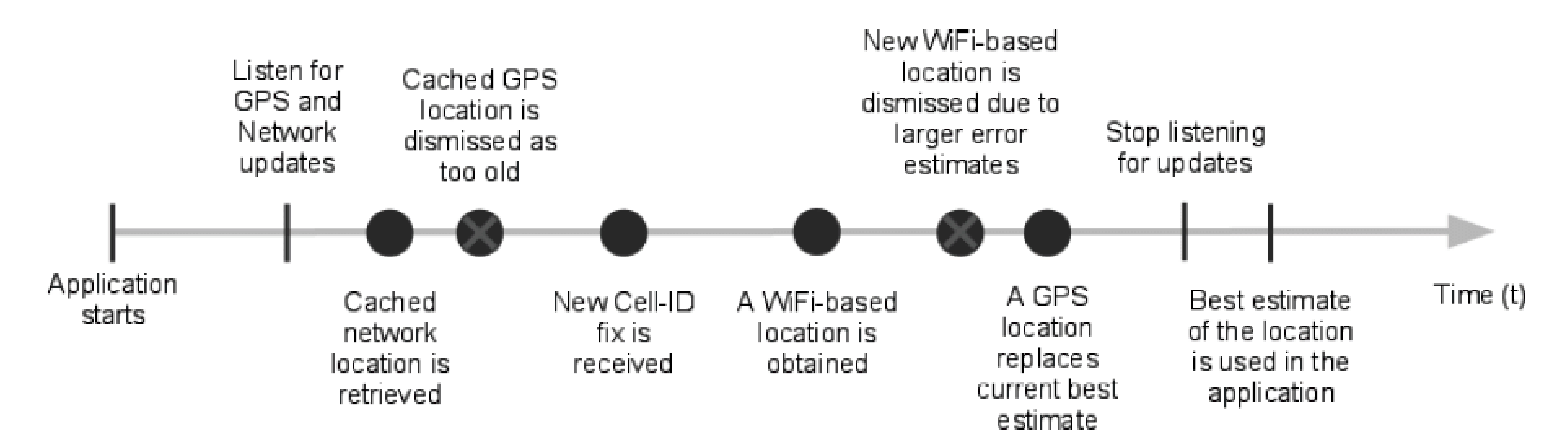


图 10.2　用户位置更新

在接收更新位置信息的这段时间,需要对以下关键点做出决策:

1) 决定开始监听更新的时刻

应用程序可以一启动就开始监听用户位置更新,也可以仅当用户触发特定的条件时才启动监听。但是要清楚地意识到两点:第一点是长时间的监听位置更新可能导致耗电量急剧上升;第二点是短时间的监听又可能使得用户位置获取的准确度不够。如上所述,可以通过调用 requestLocationUpdates()开始监听更新。代码如下:

```
LocationProvider locationProvider=LocationManager.NETWORK_PROVIDER;
//或者,使用 LocationManager.GPS_PROVIDER
locationManager.requestLocationUpdates(locationProvider, 0, 0,
locationListener);
```

2) 通过最后可知位置快速修正

位置监听器接收第一次位置更新所花费的时间长得可能让用户难以忍受。除非位置监听器接收到一个更精确的位置信息,应用程序应该暂时使用缓存中的用户位置信息,这个信息可以通过调用 getLastKnownLocation()方法来获取。

```
LocationProvider locationProvider=LocationManager.NETWORK_PROVIDER;
Location lastKnownLocation=locationManager.getLastKnownLocation
(locationProvider);
```

3) 决定停止监听更新的时刻

根据应用程序的不同,决定什么时候停止监听最新的策略可能非常简单,也可能十分复杂。在获取位置信息和使用位置信息之间加入一点时间的延迟,可能提高位置获取的准确度。持续监听会消耗大量的电量,因此只要获取了所需的信息,应该通过调用 removeUpdates()停止监听更新,代码如下。

```
//移除先前添加的监听
locationManager.removeUpdates(locationListener);
```

4) 保持最佳的估算值

最新获取的位置信息可能是最精确的。但是,由于位置修正的精确度经常变化,最新

获取到的位置信息并不一定都是最准确的。因此，需要基于一些规范添加选择位置信息的逻辑，这些规范可以根据具体的应用和现场测试的实例不同而有所变化。下面是确认位置修正可以采用的步骤：

(1) 检查是否最近得到的位置信息明显比以前的要新。

(2) 检查位置精度是好于还是差于之前的位置信息。

(3) 检查最新的位置信息是来自于哪一个提供者，并且判断这个位置信息相比之前的是否更加准确可靠。

代码 10.3 是符合上述逻辑的代码实现例子，说明了如何在应用程序中实现预定义好的策略和逻辑。

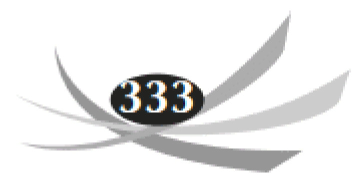
代码 10.3　位置修正判断逻辑

```
private static final int TWO_MINUTES=1000 * 60 * 2;
/** 判断哪一种位置读取方式比当前的位置修复更加准确
 * @param location 新位置
 * @param currentBestLocation 当前的位置,此位置需要和新位置进行比较
 */
protected boolean isBetterLocation(Location location, Location
currentBestLocation) {
  if(currentBestLocation==null) {
    //A new location is always better than no location
    return true;
  }

  //检查最新的位置是比较新还是比较旧
  long timeDelta=location.getTime()-currentBestLocation.getTime();
  boolean isSignificantlyNewer=timeDelta>TWO_MINUTES;
  boolean isSignificantlyOlder=timeDelta<-TWO_MINUTES;
  boolean isNewer=timeDelta>0;

  //如果当前的位置信息来源于 2min 前,使用最新位置
  //因为用户可能移动了
  if(isSignificantlyNewer) {
    return true;
  //如果最新的位置也来源于 2min 前,那么此位置会更加不准确
  } else if(isSignificantlyOlder) {
    return false;
  }

  //检查最新的位置信息是更加准确还是不准确
  int accuracyDelta=(int)(location.getAccuracy()-currentBestLocation.
  getAccuracy());
  boolean isLessAccurate=accuracyDelta>0;
```




```
  boolean isMoreAccurate=accuracyDelta<0;
  boolean isSignificantlyLessAccurate=accuracyDelta>200;

  //检查旧的位置和新的位置是否来自同一个 Provider
  boolean isFromSameProvider=isSameProvider(location.getProvider(),
      currentBestLocation.getProvider());

  //结合及时性和精确度,决定位置信息的质量
  if(isMoreAccurate) {
    return true;
  } else if(isNewer && !isLessAccurate) {
    return true;
  } else if(isNewer && !isSignificantlyLessAccurate && isFromSameProvider) {
    return true;
  }
  return false;
}

/** * 检查两个提供者是否是同一个 * /
private boolean isSameProvider(String provider1, String provider2) {
  if(provider1==null) {
   return provider2==null;
  }
  return provider1.equals(provider2);
```

5）调整模型来保存电量和数据交换

当测试应用程序的时候,可能会在模型是要提供更佳的位置信息还是更佳的效率之间做出选择调整。

6）减少窗口的大小

在一个较小的窗口下监听位置更新,意味着与 GPS 或者网络定位服务进行更少的交互,这样就可以保存电池电量。但是这样会使得可选位置变少,从而导致获取最佳位置信息变得困难。

7）减少位置提供者的更新频率

在窗口中减少更新出现的频率也可以提高电池使用效率,但是这样会牺牲精确度。两者之间的权衡要依赖于具体的实际应用。可以通过增加 requestLocationUpdates()函数的第二个和第三个参数的值来减少更新的频率。

8）仅支持一种位置信息提供者

根据应用程序的使用场景和对精度的要求,也许只需要在网络定位提供者和 GPS 之间选择一种提供者,而不是两者都需要。只和其中的一种服务进行交互可以大大减少耗电的可能性。

10.1.3 调试位置数据

在开发应用的过程中，需要对获取用户位置的模型进行效率测试。最简单的测试就是使用 Android 真机设备。但是如果没有一个真正的物理设备，也可以使用 Android 虚拟机的虚拟位置进行基于用户位置的测试。向应用提供模拟位置数据的方法主要有三种：Eclipse，DDMS 或者模拟器控制台的 geo 命令行。由于提供模拟位置数据使用的是 GPS 的数据类型，所以必须使用 GPS_PROVIDER 来获取位置更新，否则模拟数据无法工作。

如果使用 Eclipse，选择 Windows→ShowView→Other→Emulator Control。在模拟器控制面板上，在位置控制（Location Controls）下输入 GPS 坐标，GPX 文件中是路径回放，KML 文件中是多个位置的记录。确认在设备面板下已经有设备被选择，选择 Windows→Show View→Other→Devices 可以获得相关的信息。如果使用 DDMS 工具，可以使用多种方法模拟位置数据，其中包括向设备手动发送独立的经纬度；使用 GPX 文件向设备发送的一系列路径；使用 KML 文件向设备发送独立的一序列化的路径位置。如果使用模拟器控制台的 geo 命令行发送模拟位置数据，需要在 Android 模拟器上装载应用，并在 sdk 下的/tools 目录下打开设备终端的控制台，连接到模拟器控制台：

```
telnet localhost<console-port>
```

然后向模拟控制台发送位置数据。geo fix 命令发送固定的 geo 位置。这个命令接收十进制的经度和纬度，和一个可选的海拔（单位 m），例如：

```
geo fix-121.45356 46.51119 4392
```

geo nmea 发送一个 NMEA 0183 句子，例如：

```
geo nmea $ GPRMC,081836,A,3751.65,S,14507.36,E,000.0,360.0,130998,011.3,E*
62
```

10.1.4 实现位置信息获取

前面对有关定位服务的位置信息服务进行了阐述，下面利用一个简单的例子把这些知识连贯起来。该例可以在屏幕上显示手机设备当前位置的经纬度，当按下按钮时，屏幕显示出当前经纬度对应的地址信息。下面是具体的步骤。

1. 设置用户权限

在应用程序的 AndroidManifest.xml 文件中，添加设置访问位置提供器的权限内容（见代码 10.2）。

2. 定义布局等资源文件

在/res/layout 中定义用户界面的布局文件 activity_main.xml，在界面上定义一个按钮 show_address_button、一个用于响应从位置提供的信息中获取当前位置信息后，在界面上显示当前位置的经纬度以及设备所在的地址。

3. 创建或打开 Activity，获取视图对象

创建显示用户界面的 Activity 子类 ShowLocationActivity，并导入 activity_main.xml 定义的布局文件，获取按钮对象，并分别赋值给 showAddrBtn 变量。

4. 获取 LocationManager 对象

从系统获取 LocationManager 对象，并创建 Criteria 对象，根据其精度和电池耗电量的标准，使用 LocationManager 的 getBestProvider()方法选取系统中最符合要求的 LocationProvider（见代码 10.4）。

代码 10.4　初始化提供位置服务的对象

```
LocationManager  locationManager  =  (LocationManager)  getSystemService
(Context.LOCATION_SERVICE);

Criteria criteria=new Criteria();
provider=locationManager.getBestProvider(criteria, false);
Location location=locationManager.getLastKnownLocation(provider);
```

5. 定义自己的 LocationListener

在 ShowLocationActivity 中创建自己的 MyLocationListener，实现 LocationListener 的接口，当位置发生变化时，把经纬度显示在屏幕上（见代码 10.5）。

代码 10.5　自定义 LocationListener

```
private class MyLocationListener implements LocationListener {
  @Override
  public void onLocationChanged(Location location)  {
    String showLocation="Current Location \n Latitude:  "+location.
    getLatitude()+"\n Longitude:  "+location.getLongitude();
      Toast.makeText(this,showLocation ,
                Toast.LENGTH_SHORT).show();
  }

  @Override
  public void onProviderEnabled(String provider) {
      Toast.makeText(this, "Enabled new provider "+provider,
                Toast.LENGTH_SHORT).show();
  }

  @Override
  public void onProviderDisabled(String provider) {
      Toast.makeText(this, "Disabled provider "+provider,
                Toast.LENGTH_SHORT).show();
  }

  @Override
```

```
  public void onStatusChanged(String provider, int status, Bundle extras) {
    Toast.makeText(this, "Provider status changed", Toast.LENGTH_SHORT).
    show();
  }
}
```

6. 将 MyLocationLister 注册到当前的 LocationManager 对象

在 ShowLocationActivity 的 onResume()方法中注册位置监听器，并设定位置更新信息获取的方式和间隔；在 onPause()方法中删除监听器监听位置更新信息，在 ShowLocationActivity 界面处于暂停状态时减少电池耗电量(见代码 10.6)。

代码 10.6　注册监听器

```
MyLocationListener myLtn=new MyLocationListener();

@Override
protected void onResume() {
    super.onResume();
    locationManager.requestLocationUpdates(provider, 400, 1, myltn);
    }

@Override
protected void onPause() {
    super.onPause();
        locationManager.removeUpdates(myltn);
    }
```

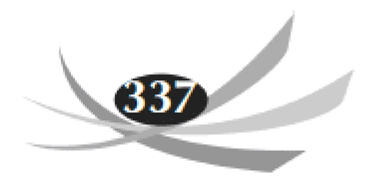
7. 定义按钮的监听器

在 ShowLocationActivity 的 onCreate()方法中使用匿名内部类的方式，定义 showAddrBtn 按钮的单击监听器，通过当前的经纬度获得确切的地址。由于实现经纬度与地址转化的代码比较烦琐，定义一个 private 的方法 getAddress()来实现(见代码 10.7)。

代码 10.7　经纬度转换为地址

```
protected void onCreate(){
    …
    showAddrBtn.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
          Toast.makeText(this, "Address: "+getAddress(),
                Toast.LENGTH_SHORT).show();
      }
    });

private String getAddress(){
```




```
    Location currentLoc=locationManager.getLastKnownLocation(provider);

    Geocoder geocoder=new Geocoder(getBaseContext(), Locale.getDefault());

    try{
        List<String>addresses=new ArrayList<String>();

      List<Address> addr=geocoder.getFromLocation(currentLoc.getLatitude(),
      currentLoc.getLongitude(), 3);

      if(addr !=null){
         for(Address address:addr){
            String placeName=address.getLocality();
            String featureName=address.getFeatureName();
            String country=address.getCountryName();
            String road=address.getThoroughfare();
            String locationInfo=String.format("\n[%s] [%s] [%s] [%s]",
            placeName,featureName,road,country);
         }
      }
      return locationInfo;
    }
    catch(Exception e){
      throw new RuntimeException(e);
}
```

运行 ShowLocationActivity，观察结果。

10.2　Google 地图

基于位置的服务(Location Based Service，LBS)是通过电信移动运营商的无线电通信网络(如 GSM 网、CDMA 网)或外部定位方式(如 GPS)获取移动终端用户的位置信息，即地理坐标或大地坐标，在 GIS 平台的支持下，为用户提供相应服务的一种增值业务。由于 LBS 与地理位置直接相关，其应用开发都离不开地图。

使用 Google Maps Android API，可以基于 Google 地图(Google Maps)的数据，在应用中添加地图功能。Google Maps Android API 可以自动处理对 Google Maps 服务器的访问、数据下载、地图显示和地图手势的事件响应。这些 API 还支持在基础地图上添加标记、多边形和图层，改变用户查看地图区域的视角。这些对象给地图位置提供额外的信息，允许用户与地图交互。

本节着重介绍 Google Maps Android API(以下简称 Maps API)中重要的类，以及如何使用 Google Maps。

10.2.1 API 中的重要类

在 API 中，Maps 是由 GoogleMap 和 MapFragment 类来描述的。使用这些 Maps API，就可以在 Android 应用程序中显示出与直接访问 Google Maps 完全相同的外观。

1. GoogleMap 类

GoogleMap 是 Maps API 的主类，是所有与地图操作相关方法的入口。当对 Map 对象进行操作时，在应用程序中根据它构建地图对象。GoogleMap 对象不能够使用构造方法直接创建，需要使用应用程序中的 MapFragment 或 MapView 的 getMap() 获得。

GoogleMap 可以自动处理下面的操作：

- 连接到 Google Maps 服务；
- 下载地图图块；
- 在设备屏幕上显示地图图块；
- 变化显示控制，例如平移和缩放；
- 响应平移和缩放手势。

2. MapFragment 类

MapFragment 类是 Fragment 类的子类，MapFragment 对象作为地图的容器，提供对 GoogleMap 对象的访问。在 Android 用户界面设计中使用 Fragment 具有很大的灵活性，可以在一个 Activity 中放置多个 Fragment，创建多个窗格的显示界面；同时，一个设计好的 Fragment 还可以在多个 Activity 中重用。

在 XML 布局资源文件中说明一个 Fragment 组件，可以使用<fragment>元素将其添加到 XML 文件中（见代码 10.8）。

代码 10.8　MapFragment 布局定义

```
<fragment
    class="com.google.android.gms.maps.MapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

MapFragment 类会自动初始化地图系统和视图，但由于初始化过程依赖于 Google Play services APK，初始化完成的时间无法确定。只有当相关的地图系统已经装载、Fragment 中相关的视图存在时，MapFragment 才能够使用 getMap()获得 GoogleMap 对象。如果无法获得 GoogleMap 对象，getMap() 方法返回空值。

当设置 ViewLifecycleInFragment()选项时，MapFragment 可以调用 onDestroyView()方法删除当前视图。只有当 MapFragment 调用 onCreateView()方法重新创建视图后，MapFragment 才有效。

如果编译目标早于 API 12 之前的应用程序，可以使用 SupportMapFragment 类获得相同的功能，但是必须在项目中包括 Android 的支持库。

3. MapView 类

MapView 类是 View 类的一个子类，允许将地图放在视图中。视图是屏幕的一个矩



形区域，是 Android 应用程序和部件的基本构建块。与 MapFragment 类似，MapView 也是作为地图容器，通过 GoogleMap 的对象呈现出地图的核心功能。

如果使用 MapView 类，必须将包含这个视图的 Activity 或 Fragment 的生命周期中所有方法的代码，在 MapView 对应的方法中同样实现。例如，一个 MapView 是一个 Activity 中的视图对象，此 MapView 则必须在 onCreate()方法中，编写与该 Activity 的 onCreate()方法中同样的代码；以此类推到生命周期中其他的方法。在 MapView 类中，下面这些方法必须要与其上层的 Activity 或 Fragment 中的代码一致：onCreate(Bundle)、onResume()、onPause()、onDestroy()、onSaveInstanceState()和 onLowMemory()。

4. Marker 类

Marker 类是用于表示地图上特定点的图标，可以称为标记。Marker 不会随着地图的旋转、分块或缩放发生改变。Marker 对象具有以下属性：

- Anchor：被放置在标记经纬度上的图片的点。默认在图片的底部，偏左位置。
- Position：地图上标记的经纬度，可以改变其值，移动标记。
- Title：用户单击标记时，显示的文本说明。
- Snippet：在 Title 下显示的额外信息。
- Icon：显示标记的位图。
- Drag Status：如果允许用户拖动标记，此属性设置为 true。
- Visibility：默认值为 true，标记可视。

代码 10.9 是一个在地图上添加标记的例子代码。

代码 10.9　添加标记

```
GoogleMap map=… //get a map.
//Add a marker at San Francisco.
Marker marker=map.addMarker(new MarkerOptions()
     .position(new LatLng(37.7750, 122.4183))
     .title("San Francisco")
     .snippet("Population: 776733"));
```

Marker 类的使用会在本章后面详述。

10.2.2　使用 Google Maps API

本节使用一个简单的例子，来说明如何在用户界面上添加地图功能。这个例子实现了在用户界面上按照某个经纬度显示标记的功能。

1. 创建 Google Maps Activity

在创建 project 时从图 10.3 左图选择 Google Map Activity，或如图 10.3 右图在 Project 中添加 Google Map Activity。本例中使用图 10.3 右图中的方法，创建一个新的 Activity，命名为 MapOneActivity。

完成 MapOneActivity 的创建后，Android Studio 自动生成了 MapOneActivity.java 和对应的布局文件，以及与 Map 服务相关的 google_maps_api.xml 文件，并在 Manifest

文件中添加相应的信息。默认情况下，Android Studio 在布局文件中简单定义一个 Fragment 来承载 GoogleMap 对象，在 MapOneActivity.java 添加获取地图 Fragement 和响应地图事件的接口 OnMapReadyCallback，见代码 10.10 和代码 10.11。

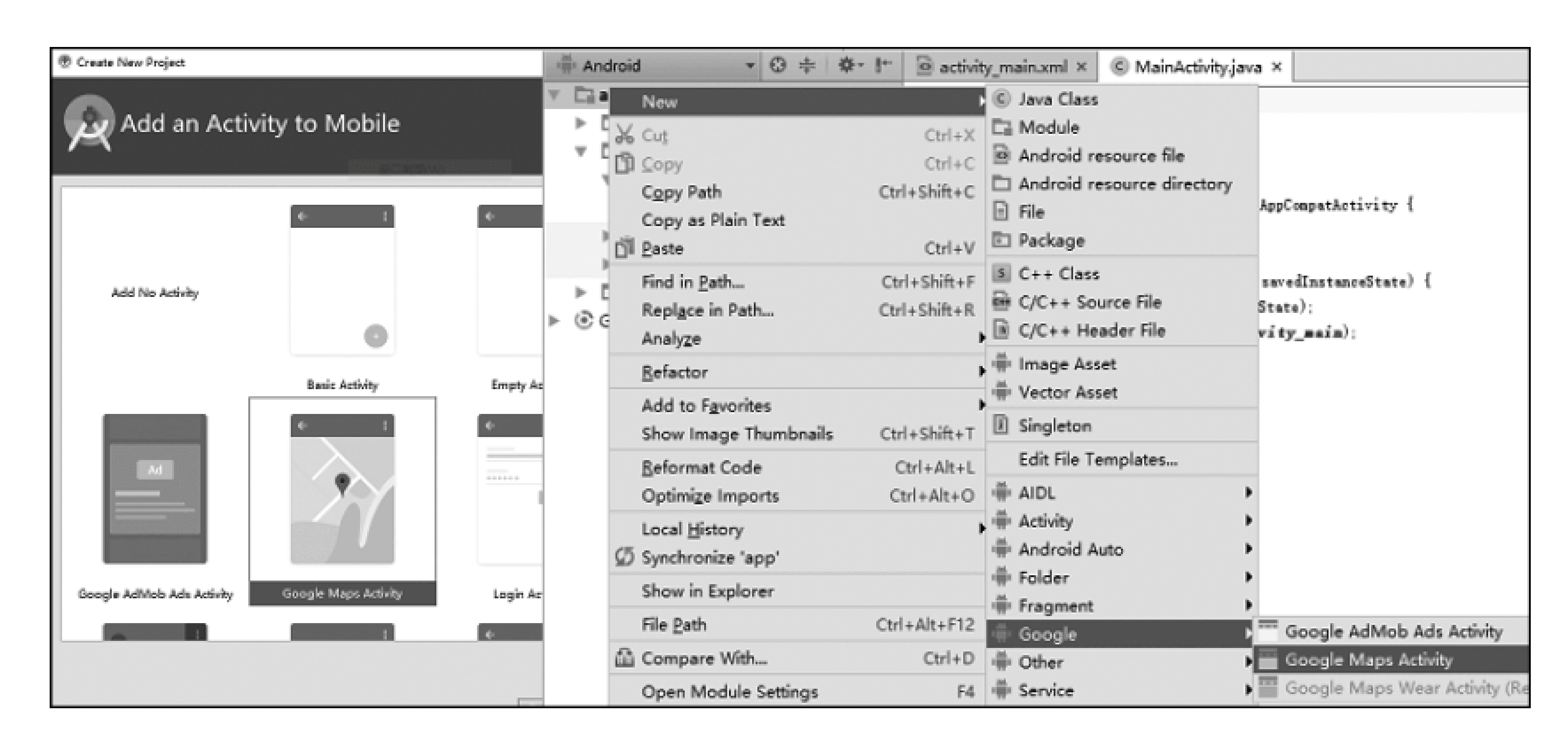


图 10.3　创建 Google Maps Activity

代码 10.10　默认布局文件

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="cn.edu.uibe.c10gmap.MapOneActivity" />
```

代码 10.11　默认 Activity 代码

```
public class MapOneActivity extends FragmentActivity
                         implements OnMapReadyCallback {

    private GoogleMap mMap;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_map_one);
        //Obtain the SupportMapFragment and get notified when the map is
        ready to be used.
        SupportMapFragment mapFragment
                  =(SupportMapFragment) getSupportFragmentManager()
                  .findFragmentById(R.id.map);
```

```
        mapFragment.getMapAsync(this);
    }

    /**…    * /
    @Override
    public void onMapReady(GoogleMap googleMap) {
        mMap=googleMap;

        //Add a marker in Sydney and move the camera
        LatLng sydney=new LatLng(-34, 151);
        mMap.addMarker(new MarkerOptions().position(sydney).title("Marker
        in Sydney"));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
    }
}
```

代码 10.11 中的 onCreate()方法中，除了获取布局文件定义的界面定义之外，还在接口 OnMapReadyCallback 的 onMapReady()方法中定义了一个位置对象，使用 Marker 在地图上进行标记。

这时，程序还不能正常运行，无法使用地图。如果要看到 Google 地图，需要从 Google 网站获取 API Key。

2. 获取 API Key

完成 GoogleMap Activity 的创建之后，要使用 Google 公司的地图资源和功能，需要从 Google 公司的网站上申请使用 GoogleMap 的密钥，也就是获取 GoogleMap API key。这一步非常重要，是获得 Google 公司地图服务的许可、能够使用 Google 地图服务的关键。值得注意的是，申请 API key 之前，首先要具有 Google 的账号，所有的操作都需要在登录 Google 之后进行。

具体的申请过程如下：

(1) 打开\res\value\google_maps_api.xml 文件，选取复制深色标注的内容，见图 10.4。

在安装 Android 的开发环境时都会有一个系统默认的证书，相当于签名，用于标记程序的开发者。选取的内容中，包含这个证书里唯一的 key，通过这个 key(MD5 认证指纹)，到 Google 的 Android Map API key 申请 App 需要的密钥。

(2) 打开浏览器，将复制的内容复制到浏览器的地址栏中，出现该 Google API 的注册界面，见图 10.5。

(3) 单击 Continue 按钮，出现界面图 10.6，单击 Create 按钮，得到所创建的本 App 的 Google API key，见图 10.7。

(4) 把图 10.7 中编辑框中的 API key 复制到图 10.4 中矩形框标注的位置，覆盖 YOUR_KEY_HERE。

到此为止，就完成了 API key 的获取，所申请的 App 获得了 Google 地图的使用权，

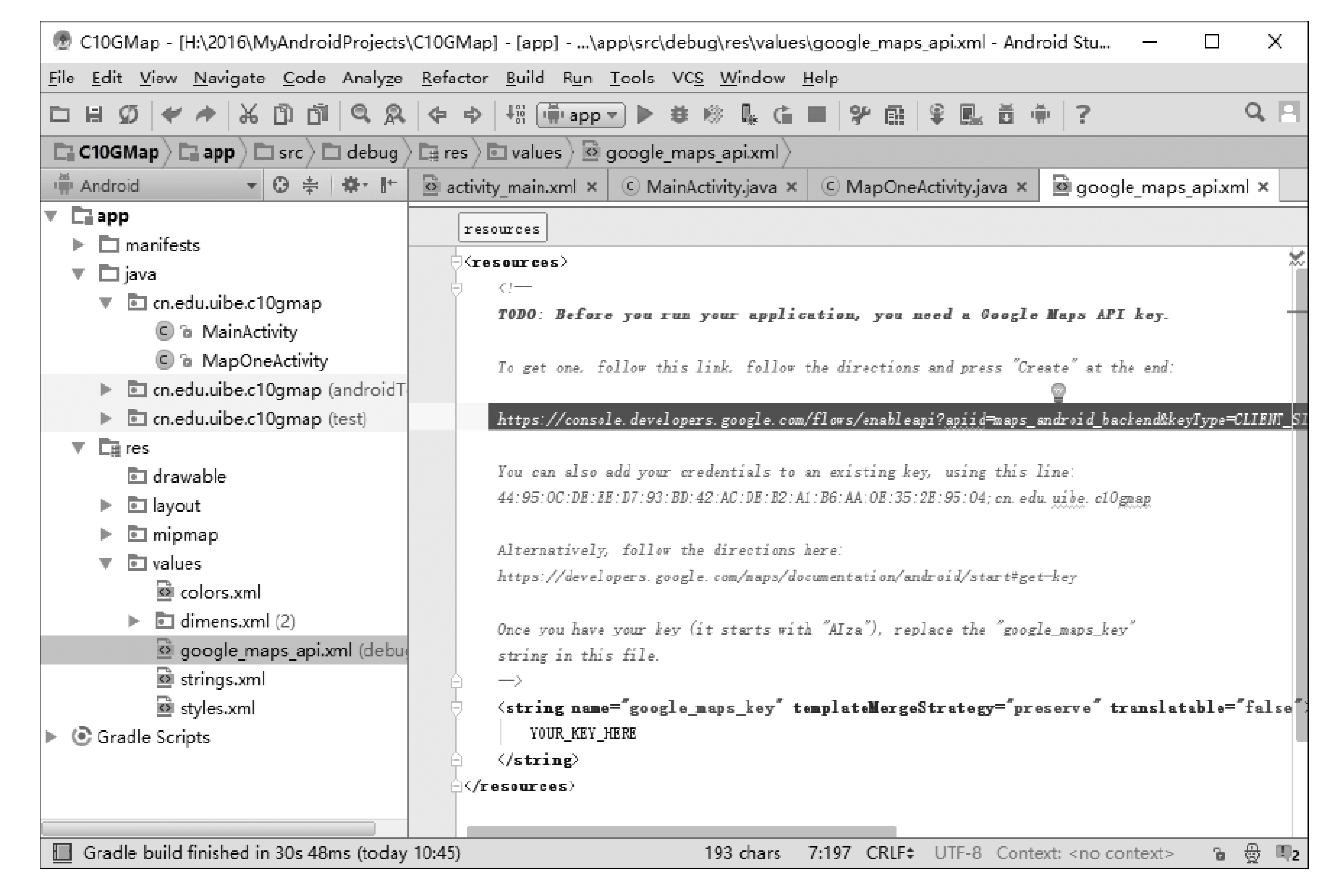


图 10.4　google_maps_api.xml 代码

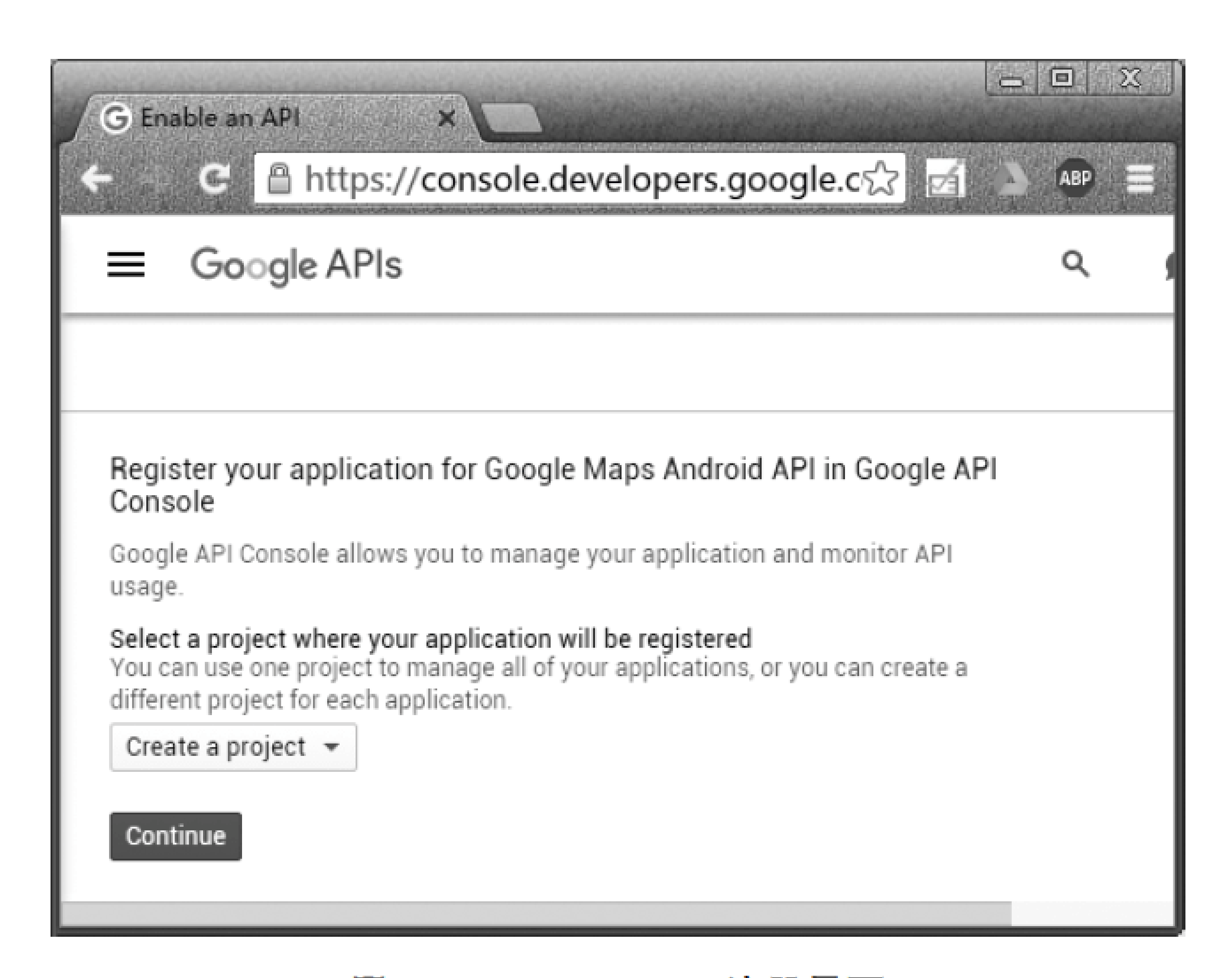


图 10.5　Google API 注册界面

可以通过网络调用 Google Map API 提供的一系列功能和资源。

运行 MapOneActivity，就能够看到悉尼的地图了。

3. 设置初始状态

在 Maps API 的应用中，可以设定地图的初始状态，以满足应用程序的需求。这其中包括摄像机的位置、缩放、方位和倾斜，地图显示类型，是否缩放按钮、罗盘显示在屏幕上，用户使用的手势等，用来操作相机。可以通过 XML 布局文件设置初始状态，也可以通过编程的方式实现。



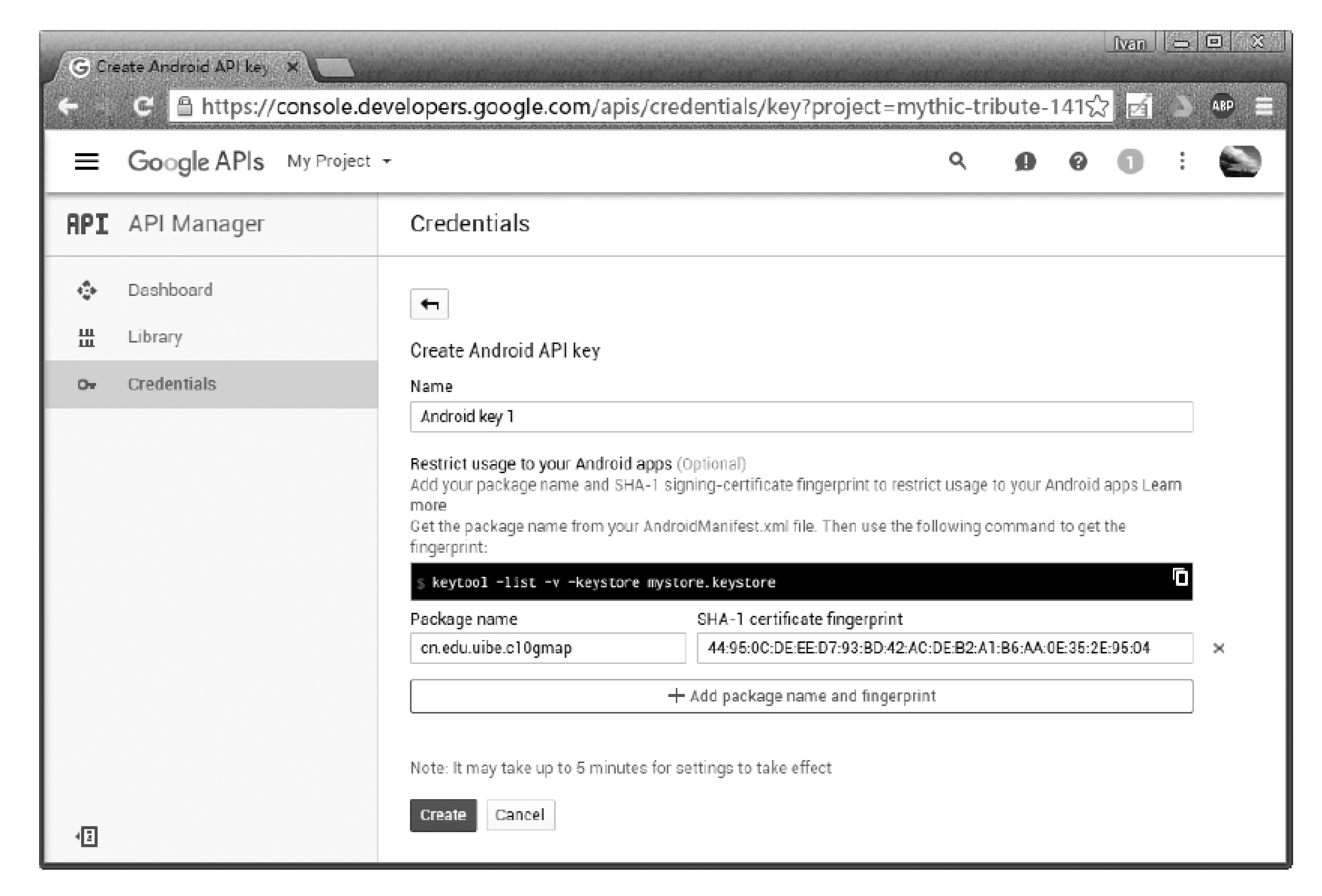


图 10.6　创建本应用的 API key

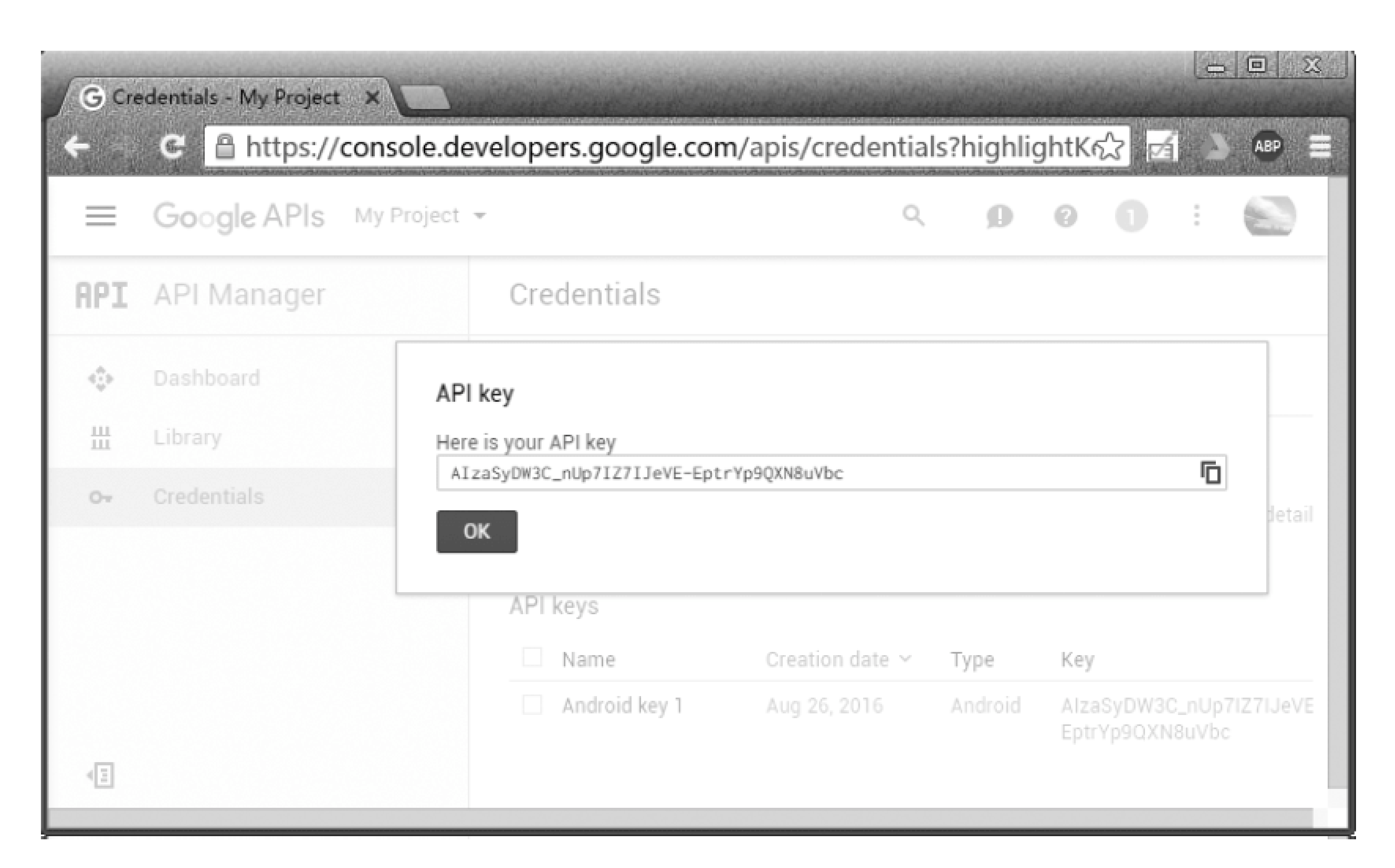


图 10.7　API key 的值

1）通过 XML 布局文件设置初始状态

如果使用 XML 布局文件添加了地图应用，地图为 MapFragment 和 MapView 定义了一组自定义的 XML 属性，可以在布局文件中直接定义初始状态。目前这些属性包括：

- mapType：可以指定地图类型的种类，其值可以是 none、normal、satellite 和 terrain。
- cameraTargetLat，cameraTargetLng，cameraZoom，cameraBearing，cameraTilt。可以指定初始摄像机的位置。

- uiZoomControls，uiCompass：可以指定是否要在地图上显示缩放控制和指南针。
- uiZoomGestures，uiScrollGestures，uiRotateGestures，uiTiltGestures：可以指定各种与地图交换手势的启用或禁用。
- zOrderOnTop：控制地图视图的表面是否被放置在其窗口的顶部。
- useViewLifecycle：只适用于 MapFragment。此属性指定是否应该将地图的生命周期连接到片段的视图或片段本身。

为了在 XML 布局文件中使用这些自定义的属性，必须首先添加下面的命名空间声明。命名空间的名称可以任何选择，不一定是 map。

```
xmlns:map="http://schemas.android.com/apk/res-auto"
```

如果命名空间的名称为 map，则使用“map:”作为前缀添加地图属性。

在上一小节例子的基础上，修改布局文件 activity_map_one.xml 中的代码，添加地图的属性设置，使地图显示时按定义的格式显示大小和模式，具体设置见代码 10.12。

代码 10.12　地图属性初始化

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="cn.edu.uibe.c10gmap.MapTwoActivity"
    map:cameraBearing="112.5"
    map:cameraTargetLat="-34"
    map:cameraTargetLng="151"
    map:cameraTilt="30"
    map:cameraZoom="10"
    map:mapType="hybrid"
    map:uiCompass="true"
    map:uiRotateGestures="true"
    map:uiScrollGestures="false"
    map:uiTiltGestures="true"
    map:uiZoomControls="true"
    map:uiZoomGestures="true" />
```

再次运行 MapOneActivity，可以看到图 10.8 右图的效果。图 10.8 右图与左图相比，显示比例增大，显示的地图类型是卫星照片和数据，显示的角度发生了改变，并具有放大和缩小按钮。这些变化都是通过在 XML 布局文件代码 10.14 一一设置的。转动图 10.8 右图中的指北针，可以调整地图的显示方位，单击右下角的放大“＋”和缩小“－”标记，可以放大和缩小地图的显示比例。

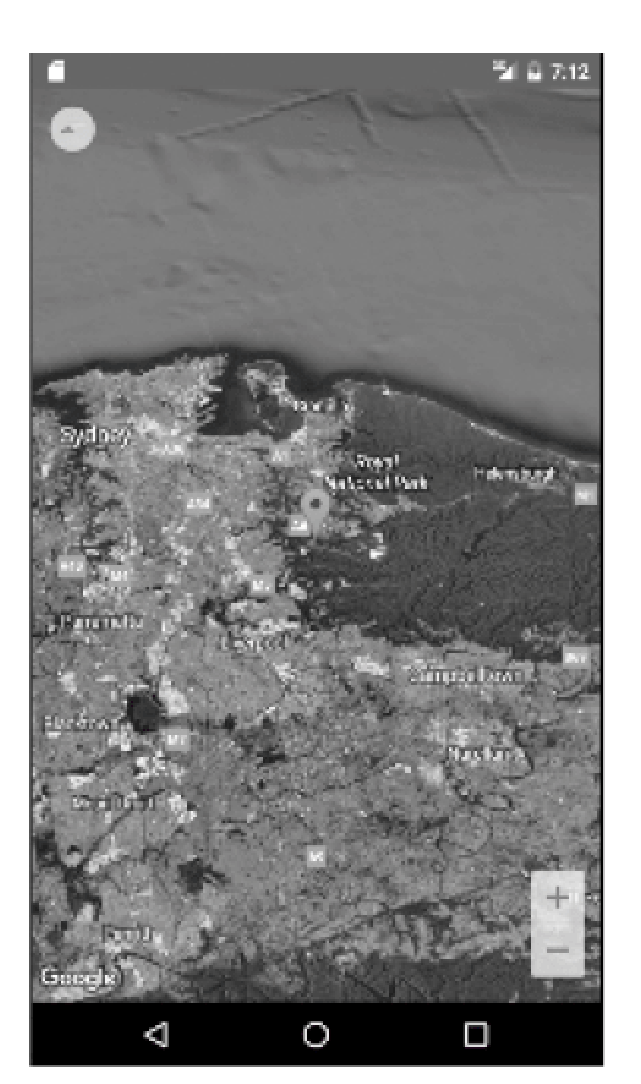



图 10.8　MapOneActiivty 运行效果

2）通过编程的方式设置初始状态

如果要通过编程的方式配置 MapFragment 或 MapView 的初始状态，需要定义 GoogleMapOptions 对象，并且使用此对象创建 MapFragment 或 MapView。首先需要创建 GoogleMapOptions 对象，代码如下：

```
GoogleMapOptions options=new GoogleMapOptions();
```

然后为 GoogleMapOptions 对象增加配置项，代码如下：

```
options.mapType(GoogleMap.MAP_TYPE_SATELLITE)
    .compassEnabled(false)
    .rotateGesturesEnabled(false)
    .tiltGesturesEnabled(false);
```

地图显示的类型，也可以通过 Java 代码来实现。在 Google Map Android API 中提供了五种类型的地图。

- MAP_TYPE_NORMAL：典型的路线图，其显示道路、某些人造的以及重要的自然特征，例如河流等。公路和特征的标签也是可见的。
- MAP_TYPE_HYBRID：在路线图上增加了卫星照片数据，公路和特征的标签也是可见的。
- MAP_TYPE_SATELLITE：卫星照片数据，公路和特征的标签是不可见的。
- MAP_TYPE_TERRAIN：地形数据。该地图包含的颜色、轮廓线和标签、透视阴影。某些道路和标签也是可见的。
- MAP_TYPE_NONE：没有图块。该地图将呈现为一个空的网格没有图块的加载。

要设置地图的类型，可以调用 GoogleMap 对象的 setMapType()方法，并且传递地图类型常量。例如，要显示卫星地图，设置代码如下：

```
GoogleMap map;
…
//Sets the map type to be "hybrid"
map.setMapType(GoogleMap.MAP_TYPE_HYBRID);
```

4. 获取 GoogleMap 对象

在 Android Studio 中，初始创建 Google Map Activity 时，平台会在 Activity 后默认实现一个接口 OnMapReadyCallback。onMapReady 是接口 OnMapReadyCallback 中需要实现的抽象方法，一旦 App 通过 onCreate()方法完成初始化，成功获取 Google 地图后，系统就会回调这个方法，对地图进行操作。在 onMapReady()方法中，可以添加标记和线条，添加监听器或移动照相机。

因为是在地图可用后系统才会调用 onMapReady()方法，所以在这个方法中可以直接通过系统传递的参数获取 GoogleMap 对象。在 10.2.2 节的 MapOneActivity 中，onMapReady()方法中通过赋值语句直接把 googleMap 赋值给成员变量 mMap，通过 mMap 的 addMarker()方法给悉尼标注了一个标记，见代码 10.13。

代码 10.13　获取 GoogleMap 对象

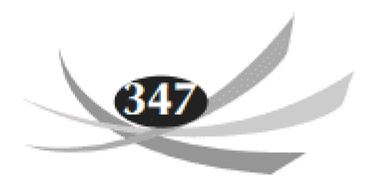
```
/**… */
    @Override
    public void onMapReady(GoogleMap googleMap) {
        mMap=googleMap;

        //Add a marker in Sydney and move the camera
        LatLng sydney=new LatLng(-34, 151);
        mMap.addMarker(new MarkerOptions().position(sydney).title("Marker
        in Sydney"));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
    }
}
```

5. 地图标记

标记可以用来识别地图上的某一位置。其使用标准的图标，与常见的 Google 地图的外观相似，可以通过 API 改变标记的颜色、图片或锚点，自定义图像。标记是 Marker 类型的对象，通过 GoogleMap.addMarker(markerOptions)方法添加到地图上。标记的图标是针对设备的屏幕绘制的，而不是在地图的表面，所以地图发生了旋转、倾斜或变焦不一定会改变标记的方向。在例子 MapOneActivity 的 onMapReady()方法中，使用 addMarker 在悉尼的经纬度上添加了红色的标记，见代码 10.11 和图 10.8。

当用户在地图上单击一个标记时，可以使用信息窗口给用户显示信息。默认情况下，如果标记有一个标题，当用户单击一个标记时，消息窗口被显示，而且同一时间只显示一



个信息窗口。如果用户单击另一个标记，当前标记的消息窗口会被隐藏，新的消息窗口将显示。可以通过调用标记的 showInfoWindow( ) 方法显示信息窗口，通过调用 hideInfoWindow()方法隐藏消息窗口。消息窗口与标记一样，是被绘制在设备屏幕上，是在标记中心的上面。默认的消息窗口包含黑体的标题，在标题下面也可以有一小段文本。

如果将标记的 draggable 属性设置为 true，则允许用户更改标记的位置；如果对标记进行长按操作，可以激活移动功能。代码 10.14 中为地图添加了一个标记，其坐标为(0，0)，当单击时在消息窗口上显示字符串“Hello world”。

代码 1.14　添加标记

```
private GoogleMap mMap;
mMap= ((MapFragment) getFragmentManager().findFragmentById(R.id.map)).
getMap();
mMap.addMarker(new MarkerOptions()
    .position(new LatLng(0, 0))
    .title("Hello world"));
```

如果改变默认的标记图像的颜色，需要传递 BitmapDescriptor 对象给 icon()方法。可以在 BitmapDescriptorFactory 对象上使用一组预定义颜色，或使用 BitmapDescriptorFactory. defaultMarker(float hue)方法设置一个自定义的标记颜色，色调的值介于 0～360，见代码 10.15。

代码 1.15　改变标记颜色

```
static final LatLng MELBOURNE=new LatLng(-37.81319, 144.96298);
Marker melbourne=mMap.addMarker(new MarkerOptions()
    .position(MELBOURNE)
    .title("Melbourne")
    .snippet("Population: 4,137,400")
    .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.
    HUE_AZURE)));
```

如果想设置更多标记的属性，而不仅仅是颜色，可以设置一个自定义标记图像，它通常被称为图标。这种自定义图标被设置为 BitmapDescriptor，可以由 BitmapDescriptorFactory 类的下面四种方法之一定义。

- fromAsset(String assetName)：创建一个自定义的标记，使用的资产目录中的图像。
- fromBitmap (Bitmapimage)：从位图图像创建一个自定义的标记。
- fromFile (Stringpath)：在指定的路径从一个文件创建一个自定义图标。
- fromResource(int resourceId)：创建一个自定义的标记，使用现有的资源。

代码 10.16 中，列出了创建一个自定义图标的代码。

代码 10.16　创建一个自定义图标

```
private static final LatLng MELBOURNE=new LatLng(-37.81319, 144.96298);
  private Marker melbourne=mMap.addMarker(new MarkerOptions()
    .position(MELBOURNE)
    .title("Melbourne")
    .snippet("Population: 4,137,400")
    .icon(BitmapDescriptorFactory.fromResource(R.drawable.arrow)));
```

在应用程序中也可以监听和响应地图标记上的事件。要监听这些事件，必须为标记在 GoogleMap 的对象上设置相应的监听器。当事件发生在地图上的一个标记时，监听器的回调方法将会被调用，并且传入相应的标记对象作为参数。在判断标记引用时，必须使用此标记的 equals()方法，而不是"=="。可以监听的事件包括标记单击事件、标记拖动事件、信息窗口的单击事件。

Maps API 针对不同的操作提供不同的监听器。监听器 OnMarkerClickListener 可以监听标记的单击事件，OnMarkerDragListener 监听标记拖动事件，OnInfoWindowClickListener 监听信息窗口的单击事件。在默认情况下，标记是不能拖动的。必须明确地设置为可拖动，标记才可以由用户拖动。

6. 调整布局文件

在 MapOneActivity 例子中，使用 MapFragment 来显示地图。如果要在 Activity 中添加其他的图形组件，需要进一步调整 activity_map_one.xml 的内容，增加布局管理器，再在布局管理器中添加图形组件和容纳地图的<fragment>元素。在<fragment>元素中，需要设置 android:name 的属性值为 com.google.android.gms.maps.SupportMapFragment，使 MapFragment 自动附加到 Activity 中。

在 MapOneActivity 例子的基础上，添加一个 TextView，显示手机当前位置的经纬度和地址信息，布局文件修改后的具体设置见代码 10.17。

代码 1.17　Fragment 设置

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="cn.edu.uibe.c10gmap.MapOneActivity">

    <TextView
        android:id="@+id/locinfo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
    <fragment xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:map="http://schemas.android.com/apk/res-auto"
```




```
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.SupportMapFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="cn.edu.uibe.c10gmap.MapOneActivity"/>
</LinearLayout>
```

除了从布局文件中设置 MapFragment 的方法，还可以通过编码的方式在 Activity 中添加一个 MapFragment 对象。使用这种方式，首选要创建一个新的 MapFragment 实例，然后调用 FragmentTransaction.add()方法将 Fragment 添加到当前的 Activity 中，代码如下：

```
mMapFragment=MapFragment.newInstance();
  FragmentTransaction fragmentTransaction=
    getFragmentManager().beginTransaction();
  fragmentTransaction.add(R.id.my_container, mMapFragment);
  fragmentTransaction.commit();
```

7. 绘制形状

可以使用 Google Maps API for Android 在地图上添加折线、多边形和圆形。折线是一系列相连的线段，可以形成任何想要的形状，可以用来标记在地图上的路径和路线；多边形和圆形都是一个封闭的形状，可以用来标记在地图上的地区。它们都有着相似的性能，并允许自定义线条的颜色、宽度等。

使用 Polyline 类在地图上定义一系列连接的线段。一个 Polyline 对象包括了一系列 LatLng 位置对象，将这些 LatLng 对象按照顺序连接起来就形成了折线。要创建一个折线，首先创建一个 PolylineOptions 的对象，然后添加位置点。位置点代表地球表面上的由经度和纬度决定的一个点，定义为一个 LatLng 对象。将这些点添加 PolylineOptions 对象中，线段是按照点与点之间的顺序绘制而成。要添加点到 PolylineOptions 对象中，需要调用 PolylineOptions.add()方法。可以连续使用这个方法一次添加多个点。添加折线的步骤为：

(1) 实例化新的 PolylineOptions 对象。

(2) 设置的 LatLng 对象，用 PolylineOptions.add()方法添加点。

(3) 根据需要设置其他属性。

(4) 调用 GoogleMap.addPolyLine 添加一个折线所指定的地图 PolylineOptions。

(5) 折线显示在地图上。

在地图上定义一个多边形的方法与定义折线是相似的，都是有很多由经度和纬度确定的位置点组成的，但多边形是封闭的，而折线不是。代码 10.18 示例了添加一个长方形在地图上的一段代码。

代码 10.18　创建形状

```
//Instantiates a new Polyline object and adds points to define a rectangle
PolylineOptions rectOptions=new PolylineOptions()
    .add(new LatLng(37.35,-122.0))
    .add(new LatLng(37.45,-122.0)) //North of the previous point, but at the
    same longitude
    .add(new LatLng(37.45,-122.2)) //Same latitude, and 30km to the west
    .add(new LatLng(37.35,-122.2)) //Same longitude, and 16km to the south
    .add(new LatLng(37.35,-122.0)); //Closes the polyline.

//Set the rectangle's color to red
rectOptions.color(Color.RED);

//Get back the mutable Polyline
Polyline polyline=myMap.addPolyline(rectOptions);
```

图 10.9 显示了代码 10.18 运行后的效果。

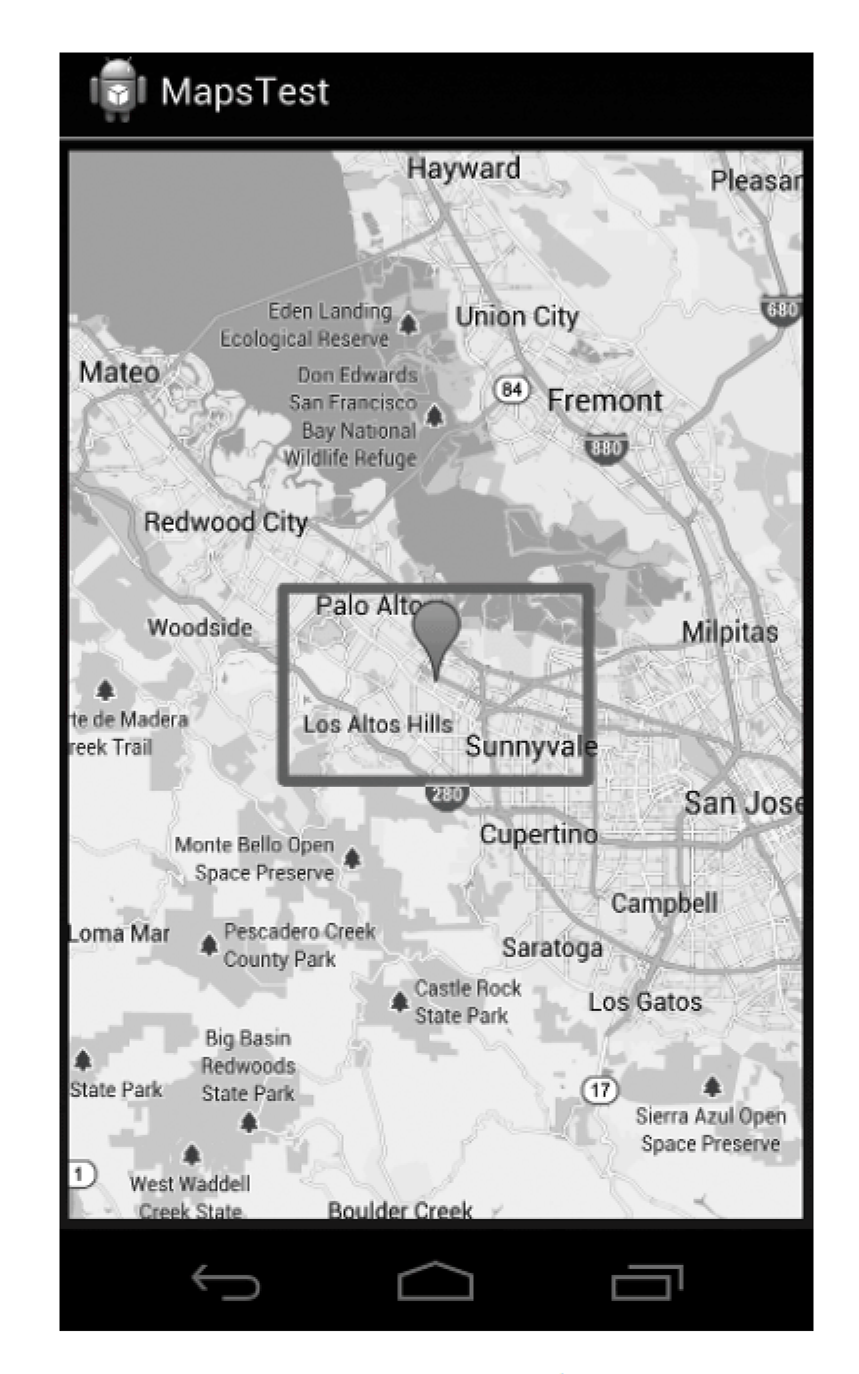
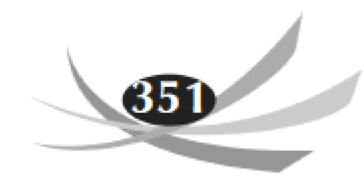


图 10.9　运行后的效果

在地图上定义圆形的过程与其他形状有所不同,需要定义两个属性:一个是中心点,使用 LatLng 表示;另一个是半径,以米为单位。

定义的圆形代表地球表面一个给定中心点和半径所确定区域的所有点集合。由于实际看到的是一个投影,如果半径比较小则看到的是几乎完美的圆形,但随着半径的增加显



示会变成非圆形。

代码 10.19 中示例了添加一个圆形的代码。首先创建一个 CircleOptions 的对象，然后调用 GoogleMap.addCircle(CircleOptions)方法。

代码 10.19　添加圆形

```
//Instantiates a new Polygon object and adds points to define a rectangle
CircleOptions circleOptions=new CircleOptions()
    .center(new LatLng(37.4,-122.1))
    .radius(1000)); //In meters

//Get back the mutable Circle
Circle circle=myMap.addCircle(circleOptions);
```

圆形被添加后，如果要改变其形状，可以调用 Circle.setRadius()或 Circle.setCenter()方法。

10.3　本章小结

本章主要介绍了 Android 应用程序如何使用实现定位服务和地图服务。

Android 通过 android.location 包中的类为应用程序提供定位服务。定位框架中的核心组件就是 LocationManager 系统服务，其提供了支撑底层设备的定位 API。Android 设备可以使用 GPS 和 Android 网络位置提供器(Android Network Location Provider，NLP)来提供位置信息，提供与位置相关的服务。

Android 应用程序通过调用 Google Maps Android API(简称 Maps API)来实现基于 Google Maps 的地图服务。Maps API 可以自动处理对 Google Maps 服务器的访问、数据下载、地图显示和地图手势的事件响应。在 Maps API 中，常用的重要类包括 GoogleMap、MapFragment、MapView 和 Marker。Android 应用程序在使用 Maps API 之前，需要安装 Google Play services SDK，获取 Maps API 密钥，并进行 menifest 文件中的密钥和权限设置。

附录 A
Eclipse 的 Android App 开发环境

A.1　Android 开发环境搭建

在进行 Android 应用程序开发之前，需要搭建 Android 应用程序开发环境。在本书中，采用开源的 Java 集成开发环境 Eclipse 作为开发工具，并添加必需的 Android SDK 和其他插件。

对于新开发者来说，Android 的官方网站 http://developer.android.com 提供了包含 Android SDK 的 Elipse 安装包链接，开发者可以直接从链接下载支持 Android 系统的 Eclipse 安装包。但这里还分别介绍了安装各包的步骤，以便开发者学习。按照 Android 开发环境的安装顺序，可以将这个过程分为四大步骤：

- 安装 Eclipse 开发环境；
- 安装 Android SDK；
- 安装 Eclipse ADT 插件；
- 安装 Google Play services SDK。

如果已经熟悉 Eclipse 开发环境的安装，可以跳过 A.1 节下面的相关内容，直接阅读 A.2 节及后面的内容。

A.1.1　安装 Eclipse 开发环境

Eclispe 开发环境的安装包括安装 JDK、下载和安装 Eclipse 以及配置 Eclipse 三个部分。

1)安装 JDK

从 SUN 公司发布的 JDK 的官方网站下载 Windows 版本的 JDK 安装软件，最新版本为 JDK1.7。

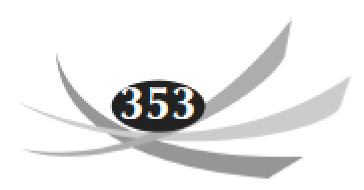
下载后选定所安装的磁盘和目录，判断磁盘空间是否够用。默认状态下 JDK 安装在启动盘的 C:\Program Files\Java 目录下。

运行 JDK 安装包，根据安装向导完成 JDK 的安装。安装完成后，在 Windows 中进入行命令 cmd 窗口，测试安装结果。

2) 下载和安装 Eclipse

从 Eclipse 官方网站(http://www.eclipse.org/downloads/)下载最新版本的 Eclipse。请选择 Eclipse IDEforJavaEEDevelopers，目前的版本是 3.7.1。

下载 Eclipse 后，将压缩包直接解压到硬盘上，会新建一个 Eclipse 目录存放 Eclipse



的文件。假设解压缩到 D 盘,则 Eclipse 安装在 D:\eclipse 目录下。

3）配置 Eclipse

进入 Eclipse,首先看到欢迎界面。关闭欢迎界面,可以看到默认状态下 Eclipse 平台的各个视图,见图 A.1。

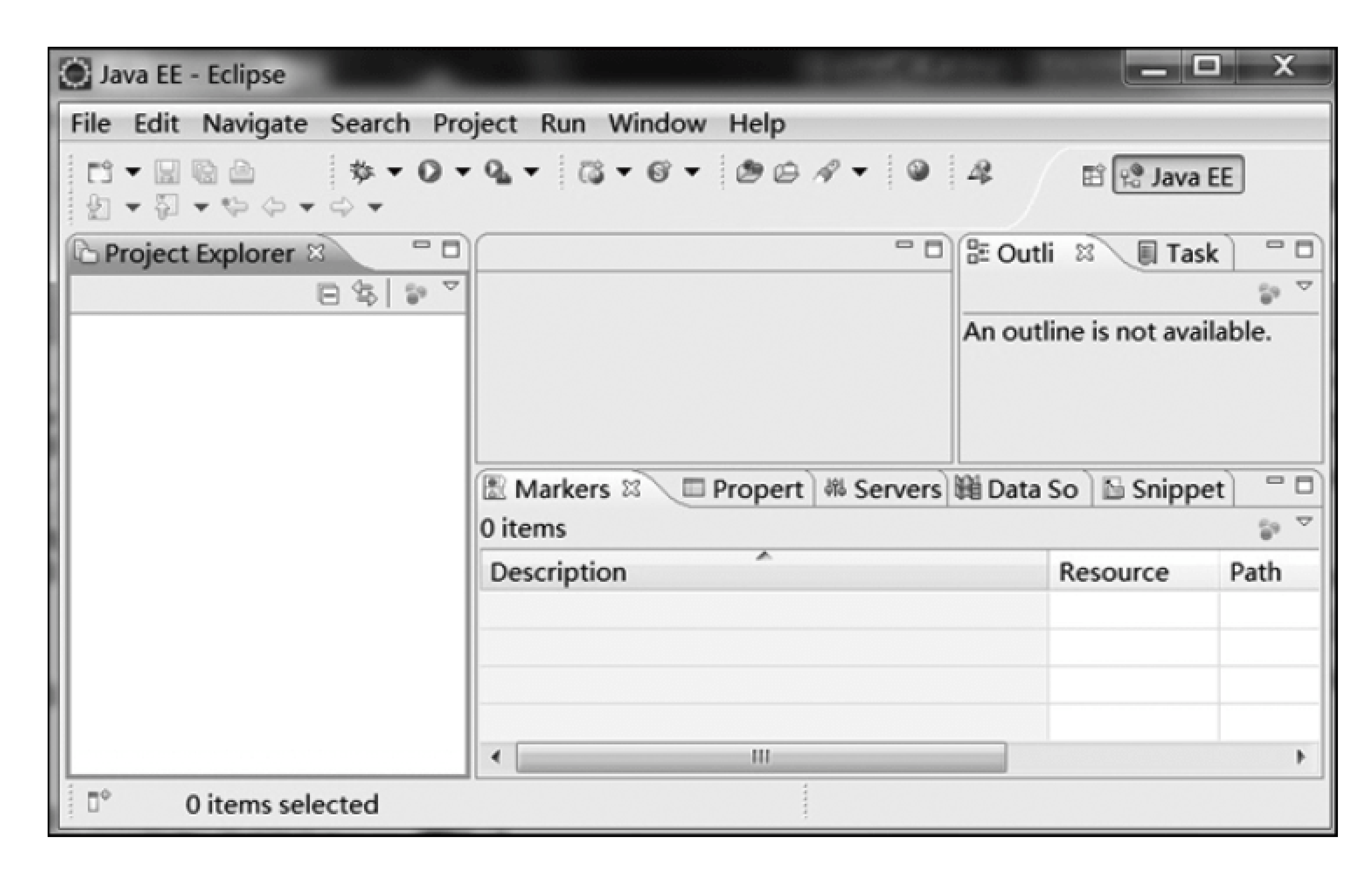


图 A.1　Eclipse 平台视图

选择 Window→Preferences,打开 Preferences 对话框查看 Eclipse 的选项,检查 JRE 的安装是否正确(见图 A.2),同时学习使用其他配置项。

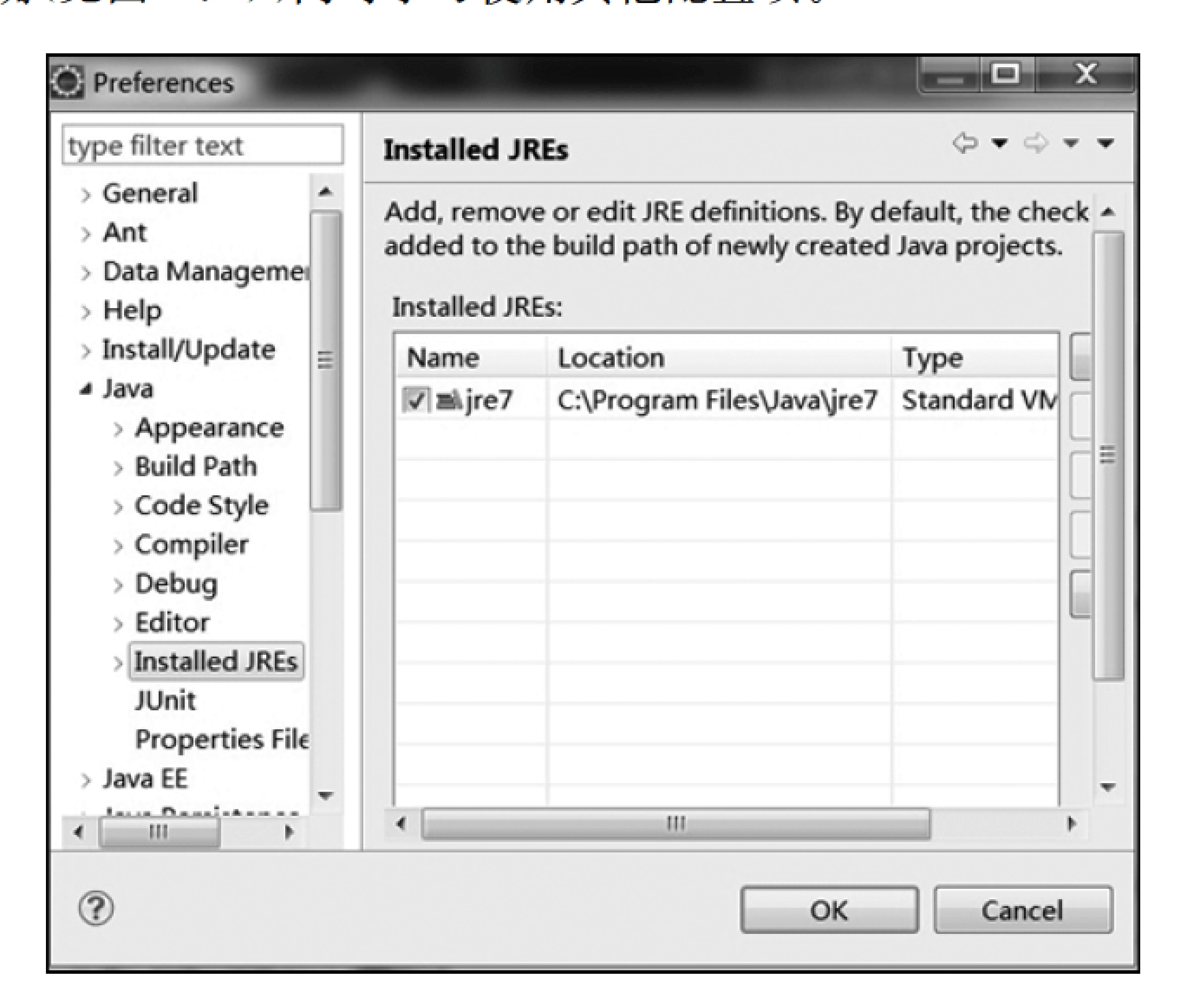


图 A.2　检查 JRE 的安装

A.1.2　安装 Android SDK

Eclipse 安装完成后,进行 Android SDK 安装,分两个步骤。

（1）下载 Android SDK 安装程序。

从 http://developer.android.com/sdk/index.html 下载最新的安装程序，并安装。

（2）添加平台和其他组件。

使用 Android SDK Manager（该工具包含在 SDK starter 包中）来下载必要的 SDK 组件到开发环境中。

如果使用的是 Windows 安装程序，它会在安装结束后自动运行 Android SDK Manager（见图 A.3），只需要接受推荐的组件集并安装就可以了。当然也可以手动运行 Android SDK Manager。在 Linux 系统中，打开终端并进入/tools 目录，然后执行 Android，将弹出图 A.3 所示界面，在界面中可以浏览 SDK repository 并选择新的或更新的组件。

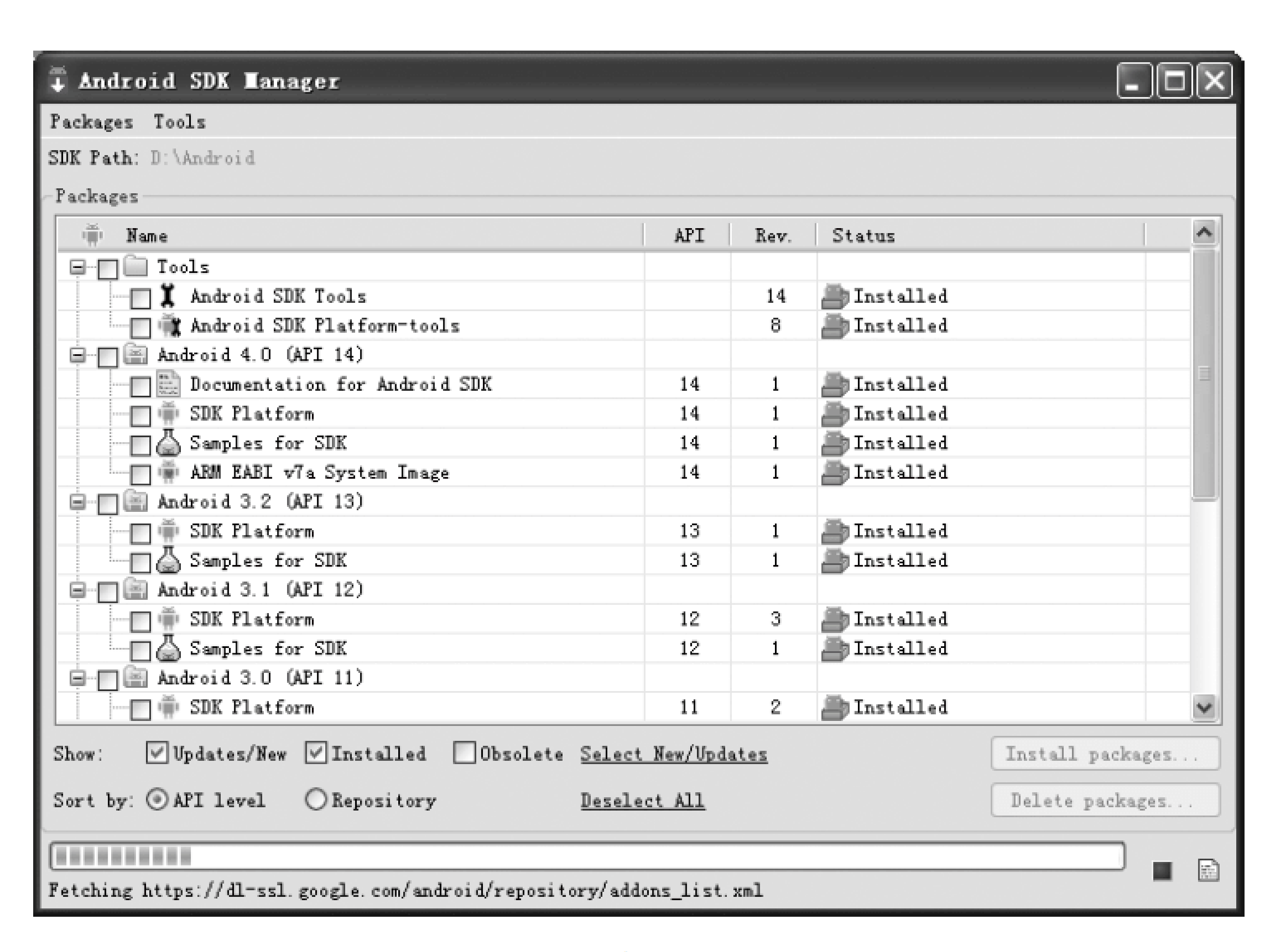


图 A.3 下载 Android SDK

A.1.3 安装 Eclipse ADT 插件

在 Eclipse 环境中编写 Android 应用程序，通过 Android SDK 的调用，可以直接使用 Eclipse 进行编译。但 Android 应用程序最终运行在手机上，因此需要一个手机的模拟运行环境来测试 Android 应用程序的运行状况。安装 Eclipse ADT 插件，可以使 Eclipse 启动智能手机的模拟器，完成在 Eclipse 上的 Android 应用程序模拟运行测试。

Eclipse ADT 插件可以在线安装，也可以下载到本地手动安装。

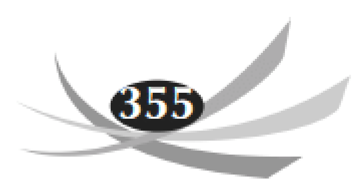
1）下载 ADT 插件

（1）启动 Eclipse，选择 Help→Install New Software。

（2）单击右上角的 Add 按钮。

（3）在 Add Repository 对话框中，输入名称 ADT Plugin 以及如下地址：

```
https://dl-ssl.google.com/android/eclipse/
```



注意：如果获取插件有问题，使用 http 协议代替 https。

（4）在 Available Software 对话框中，选中 Developer Tools 复选框并单击 Next 按钮。

（5）在下一个窗口中，将看到一系列可下载的工具，单击 Next 按钮。

（6）阅读并接受协议，单击 Finish 按钮。

（7）安装完成后重启 Eclipse。

2）配置 ADT 插件

（1）选择 Window→Preferences。

（2）在左侧面板中选择 Android。

（3）在主面板的 SDK Location 中单击 Browse 按钮，并定位到所下载的 SDK 目录。

（4）单击 Apply 按钮，然后单击 OK 按钮。

3）下载本地并手动安装

如果无法使用 Eclipse 下载 ADT 插件，可以下载 ADT 的 zip 文件到本地并手动安装。

A.1.4　安装 Google Play services SDK

如果在 Android 应用系统中需要用到地图等功能，还需要安装 Google Play services SDK。Google Play services SDK 由 Android SDK 管理器安装。

（1）启动 SDK 管理器。从 Eclipse 启动 Android 的 SDK 管理器。

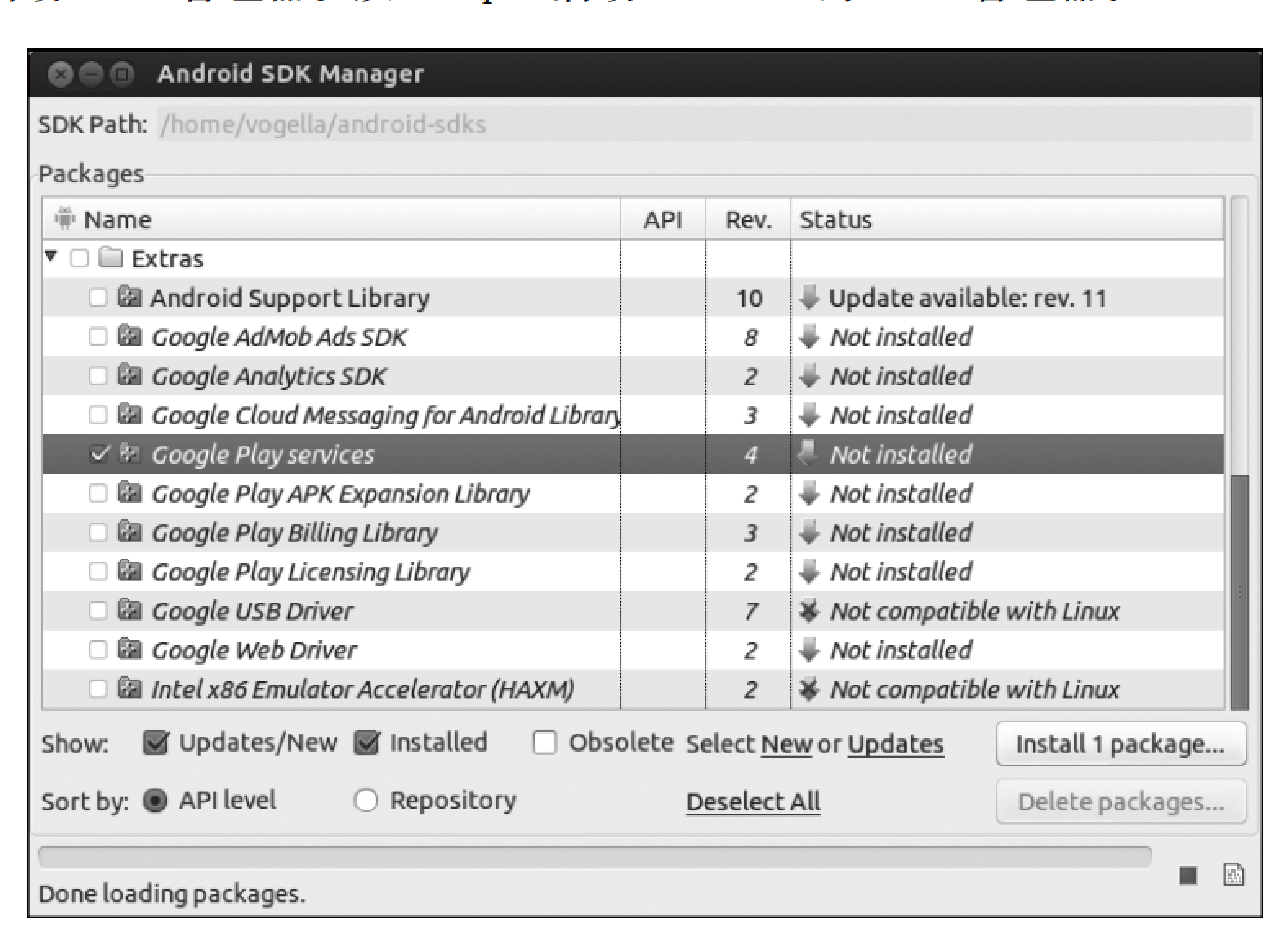


图 A.4　下载安装 Google Play services SDK

（2）滚动到软件包列表的底部，选择 Extras→Google Play services，并安装它。

（3）在使用 Google Play services 的相关功能时，将目录＜android-sdk-folder＞/extras/google/google_play_services/libproject/google-play-services_lib 复制到 Android

应用程序项目下，具体的操作过程会在后续节使用时详细说明。

A.2 第一个 Android 应用程序

在完成了 Android 开发环境的安装和配置后，就可以使用 Eclipse 开始 Android 应用程序了。下面介绍第一个 Android 应用程序。Hello Mobile World 的开发和运行过程，它的功能是在界面上显示“您好，移动世界”的字符。

第一次编写 Android 应用程序需要完成四个步骤：

步骤一：创建 AVD。

步骤二：创建一个新的 Android 项目。

步骤三：创建用户界面。

步骤四：运行应用程序。

A.2.1 创建 AVD

因为 Eclipse 运行 Android 应用程序时，是在 Android 仿真器中运行。在编写和运行 Android 应用程序之前，第一步必须先创建一个 Android 虚拟设备，也就是 Android Virtual Device(AVD)。AVD 定义了系统镜像以及仿真器需要的设备设置，模拟了真实 Android 系统下的环境。

具体的操作步骤如下：

(1) 在 Eclipse 中，选择 Window→Android SDK and AVD Manager。

(2) 在左侧面板中选择 Virtual Devices。

(3) 单击 New 按钮，出现 Create new Android Virtual Device 对话框(见图 A.5)，在对话框中对 AVD 的名称、软件环境和硬件属性进行配置。

在 Name 文本框中输入新的 AVD 的名称，如“my_avd”；在 Target 下拉列表框中选择 AVD 所使用的 Android SDK 版本。如果在 Android SDK 安装过程中安装了几个版本，都可以从这里看到；CPU/ABI 指示的是模拟器运行时模拟的 CPU 型号；SD Card 的 Size 输入框可以设定模拟器的内存大小，例如设置为 512MB；快照 Snapshot、模拟器的皮肤 Skin 和硬件配置 Hardware 可以采用默认值。

配置完成后，单击对话框下方的 Create AVD 按钮，完成 AVD 的创建过程。

(4) 选择一个目标。目标即是想在仿真器上运行的平台(Android SDK 的版本号，如 2.1)。可以忽略剩下的文本框。

(5) 单击 Create AVD 按钮，这时从左面列表中，就可以看到创建好的 AVD。

AVD 前面的复选框处于选中状态，表示这个 AVD 可以正常运行，通过右边的按钮，可以对创建好的 AVD 进行编辑、修复等操作，单击 Start 按钮可以直接启动 AVD。可以创建多个配置不同的 AVD，在列表中单击相应的 AVD 后，直接单击 Start 按钮运行该 AVD。

(6) 关闭 AVD Manager 对话框，回到 Eclipse 主界面。

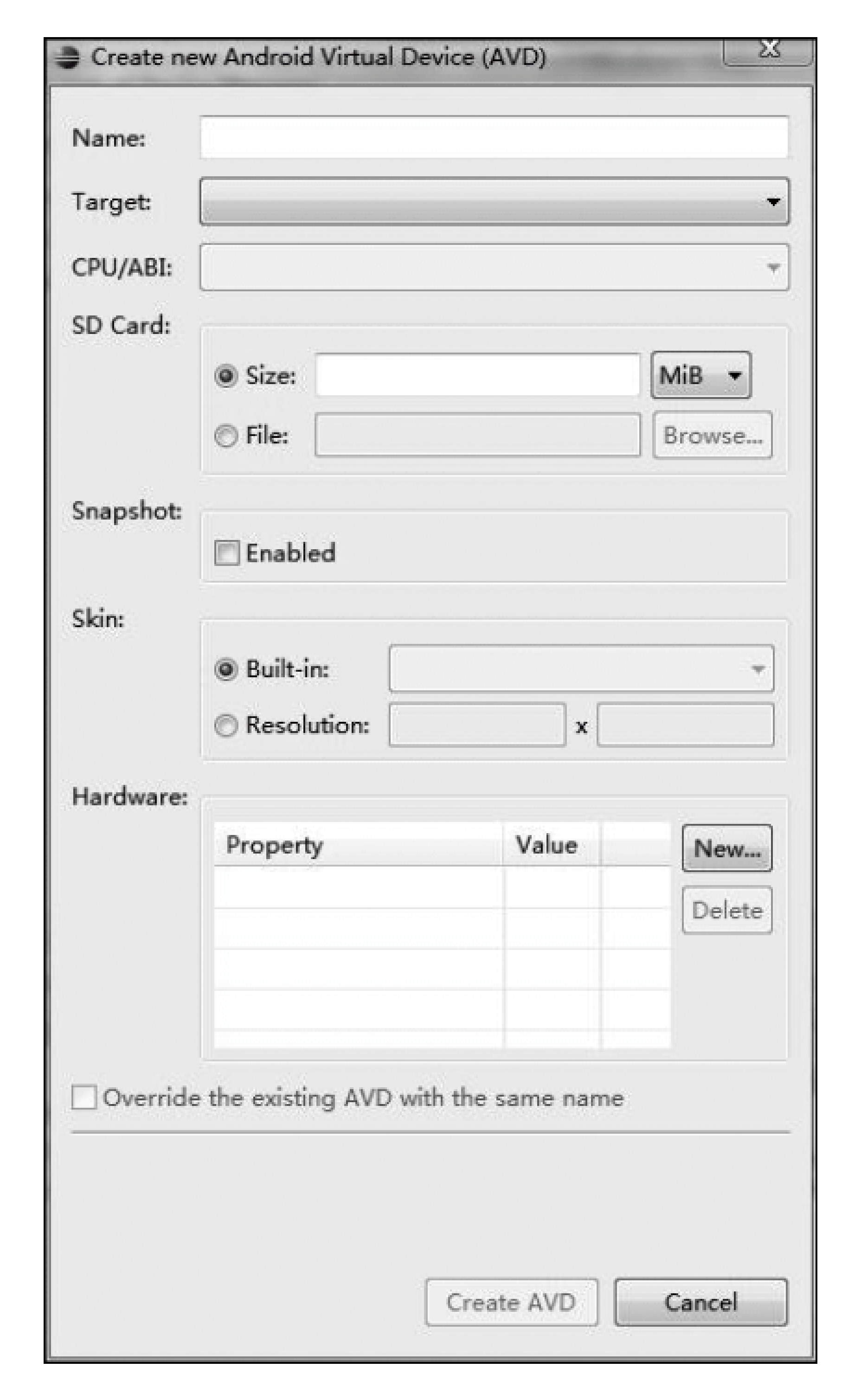


图 A.5　Create new Android Virtual Device 对话框

A.2.2　创建一个新的 Android 项目

AVD 创建完成后，可以按照创建 Eclipse 的 Java 或 Web 项目等类似的步骤，在 Eclipse 中创建一个 Android 项目。

具体的操作步骤如下：

(1) 在 Eclipse 中，选择 File→New→Project。如果已经成功安装 ADT 插件，对话框中将出现一个名为 Android 的文件夹，该文件夹中包含 Android Project(在创建了一个或多个 Android 项目后，条目 Android XML File 也将变成可用状态)。

(2) 选择 Android Project，然后单击 Next 按钮。

(3) 在对话框中填写如下内容：

- Project name：HelloWorld。
- Application name：Hello，Mobile World!。
- Package name：cn.edu.uibe.mc.sample(或者你自己的私有命名空间)。
- Create Activity：HelloWorldActivity。

Project name 是项目名称，Eclipse 在工作区中创建与项目名相同的目录名，该目录中包含项目文件。

Application name 是应用程序的名称应用程序的标题。这个名称将作为应用程序的标题，显示在 Android 设备上，这是给用户的提示信息。这里给该应用取名为 Hello

Mobile World!。

Package name 是 Java 包的名称,这与标准 Java 语言中包的概念相同。在此包中,将生成主 Activity。这个应用的其他类也放在 cn. edu. uibe. mc. sample 包及其子包中。

Create Activity 是主 Activity 的类名称,是 Activity 类的子类。一个 Activity 就是一个普通的类,它能创建一个 Android 用户界面。一个 Activity 就是一个普通的类,它能创建一个用户界面,当然这并不是必需的。由于有复选框,因此创建 Activity 是可选的,但是 Activity 通常被作为一个应用程序的基础。这里输入类名称 HelloWorldActivity。

Min SDK Version 值定义了应用程序所需要的最小 API 等级。更多信息请参考 Android API Levels。

Use default location 复选框允许改变即将生成的项目文件在磁盘上的路径。

Build Target 是应用程序将被编译的平台目标(基于 Min SDK Version,该值将被自动设置)。

注意:假如已经选择了 Android 1.1 平台作为 Build Target。那么意味着应用程序将在 Android 1.1 平台库基础上进行编译。如果之前创建的 AVD 运行在 Android 1.5 平台上,那么没关系,Android 应用程序是向前兼容的,因此在 1.1 平台库上构建的应用程序可以正常运行在 1.5 平台上,反之则不行。

(4) 单击 Finish 按钮。

完成 Android 项目的创建之后,就可以在这个项目中对 Android 用户界面进行定义和修改了。

从左面的 Package Explorer 视图中,可以看到前面所建的 Hello 项目,单击打开,查看 src 目录下的 cn. edu. uibe. mc. sample 下的 Java 源程序 HelloWorldActivity. java。打开该 Java 源文件,其内容见代码 A. 1。

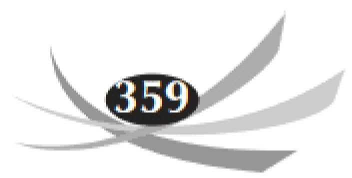
代码 A.1 HelloWorldActivity. java

```
import android.app.Activity;
import android.os.Bundle;

public class HelloWorldActivity extends Activity {
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
                                              }
```

从代码 A. 1 中可以看出 HelloWorldActivity 是 Activity 的子类。Activity 是 Android 系统中用于实现用户图形界面的类。

public void onCreate()方法是 Activity 所定义的方法,当 Activity 启动时由 Android 系统调用,会首先在这个方法中执行初始化和用户界面设置工作。



一个 Activity 并不一定要有用户界面,但通常都会有。Activity 概念可以参照 Java 中的 Applet 概念来理解。

A.2.3　创建用户界面

如果要在 Android 界面上显示"您好,移动世界",需要对自动生成的 HelloWorldActivity.java 代码进行修改,在 onCreate()方法中添加显示字符串的组件,并设置其显示出来,见代码 A.2。

代码 A.2　修改后 HelloWorldActivity.java

```
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloWorldActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv=new TextView(this);
        tv.setText("您好,移动世界!");
        setContentView(tv);
    }
}
```

在 Java 程序导入包的时候,可以直接使用快捷键 Ctrl+Shift+O。

在代码 A.2 中定义了一个文本显示框 TextView 的对象 tv,将 TextView 类对象作为 Activity 用户界面显示的内容,传给 setContentView()方法。Activity 通过调用 setContentView(tv),把 tv 的内容显示出来。如果 Activity 不调用 setContentView()方法,则不会显示任何用户界面,系统会显示一片空白。

下一步,就是运行应用程序了。

A.2.4　运行应用程序

在仿真器中运行 HelloWorldActivity 的步骤如下:

(1) 在 Eclipse 中,选择 Run→Run。

(2) 选择 Android Application,Eclipse 插件自动为应用程序创建一个新的运行配置并启动 Android 仿真器。

(3) 当仿真器启动后,Eclipse 插件将安装应用程序并运行默认的 Activity。

启动仿真器后,可以看到一个手机的界面。左面是显示屏幕,右边是手机的键盘按键,通过键盘的按钮可以直接对仿真器进行操作。

这时需要真正耐心等待运行结果。一般来说,第一次在仿真器上运行 Android 应用程序需要花更长的时间。具体的时间与机器的性能和配置有关。

如果 5～6min 都只看到仿真器上的 logo 在闪烁，不要着急，去休息一下，过一会儿再来。

图 A.6 是修改后 HelloWorldActivity.java 的运行结果。上面一行文字是前面定义的 Application name；下面一行是 TextView 对象显示的内容。

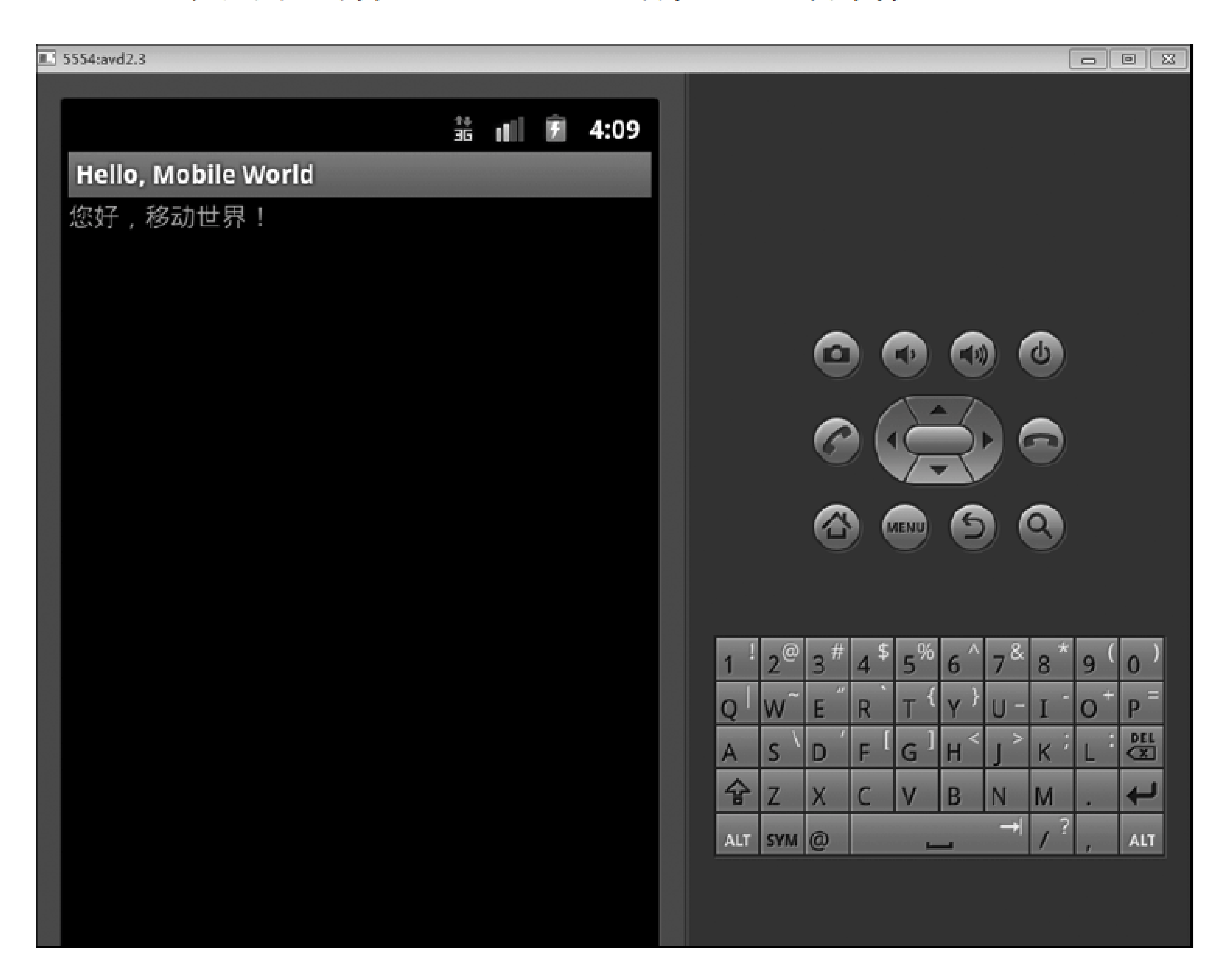


图 A.6　AVD 运行结果

单击右面键盘上的返回键，返回 Android 系统的主界面，见图 A.7。



图 A.7　Android 系统的主界面

这是 Android 系统的主界面，除了刚才编写的程序外，还有一些其他的程序，例如时钟、日历、照相机、图片等，这些程序是创建 Android 项目时，由 Android SDK 加载的 Demo 程序。

选择任意一个 logo，如选择 API Demos→animation，启动 Android 系统的应用程序，查看结果。

单击右面键盘上的 Home 键，回到主界面，单击图 A.7 中左下角的白底绿小机器人（这是应用程序的 logo），可以重新看到图 A.6 的显示结果。

图 A.6 中的"Hello Mobile World"的界面中，灰色条显示的"Hello，Mobile World"就是应用程序标题，该字符串定义在 res/values/strings. xml 文件中并被 AndroidManifest. xml 文件所引用，标题之下的文本就是 Java 源程序在 TextView 对象中设置的。

这里使用的是编程式的用户布局方式，也就是使用 Java 程序来定义用户界面上的 View 图形组件。这种布局方式有很大的缺点，如一些小的布局变化有可能导致大的源码修改，编写程序时也很容易忘记将多个 View 链接在一起。

以上就是构建"Hello，Mobile World"应用程序的过程，但这不是 Android 系统推荐的设计用户界面的方法，Android 提供了一个可选的用户界面构建模型：基于 XML 的布局文件的用户界面构建模型。

下面介绍如何使用 XML 布局文件来构建用户界面。

A.2.5　使用 XML 来定义用户界面

刚才完成的"Hello，Mobile World"应用程序使用的是编程式的用户布局方式。使用这种方式是直接在源代码中构建应用程序的用户界面。这种方式对于界面的开发是很不方便的，因为一些小的布局变化都有可能导致源码的修改，并且需要重新编译，所以我们需要使用 MVC 开发模式。Android 也设计了一套 MVC 开发模式，其提供了一个基于 XML 的布局文件来定义用户界面。

Android 系统推荐使用 XML 布局文件来设计用户界面。XML 布局文件都位于应用程序的 res/layout/目录中。res 是 resources 的缩写，该目录中包含应用程序所需要的所有非代码资源。除了布局文件之外，资源还包括图片、声音以及本地字符串。

在 Package Explorer 中，展开 res/layout/文件夹。在创建 Android 项目时，Eclipse 插件自动创建一个布局文件：main. xml。在前面修改完成的"Hello Mobile World"应用中，该文件被忽略了，而以编程的方式创建了布局。这是为了介绍更多关于 Android 框架知识。通常情况下，建议使用 XML 文件布局而不是硬编码。

双击打开 main. xml 文件，单击 Java 视图里的 main. xml 小标签，XML 文件内容见代码 A.3。

代码 A.3　main. xml

```
<?xml version="1.0" encoding="utf-8"?>
   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
```

```
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical">

        <TextView
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/hello" />

</LinearLayout>
```

一个 Android XML 布局文件的总体结构比较简单：就是一棵 XML 元素的树，树中的每个节点都使用 View 类的类名作为元素名称，每一个元素都对应一个用户图形界面中的组件，在代码 A.3 中只有一个元素 TextView。

XML 布局中的元素可以是任何 View 子类，包括自定义的 View 类。与编程布局方式相比，XML 布局使用了更简单的结构和语法，能够快速构建用户界面。这种模型的灵感源于 Web 开发模型，使用这种方式，把应用程序的用户界面从应用逻辑中分离了出来。仔细查看代码 A.3 内容，初步了解 XML 布局的语法格式。

＜?xml version＝"1.0" encoding＝"utf-8"?＞是版本和采用的编码标准，这是布局类型定义。在这个 XML 布局文件中，只定义了一个 View 元素：TextView，它有 4 个 XML 属性，分别在下面进行了定义：

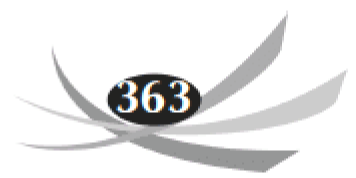
- xmlns:android。这是 XML 命名空间声明，它告诉 Android 工具应用程序将要引用在 Android 命名空间中定义的普通属性。在每个 Android 布局文件的最外层标签中必须包含这个属性。
- android:layout_width。定义了 TextView 在屏幕上会占用的可用宽度。在本例中，由于只有一个 View，这里设置它占据整个屏幕，因此定义了值 fill_parent。
- android:layout_height。定义了 TextView 在屏幕上会占用的可用高度。属性值和 android:layout_width 相似。
- android:text。设置了 TextView 将要显示的值。在本例中，使用了字符串资源而不是直接给出了字符串值。字符串定义在 res/values/strings.xml 文件中。这是一种推荐的方式，因为这可以使应用程序能够很好地本地化，而不需要改变布局文件。

下面修改程序代码，使这个 hello 程序采用 XML 布局文件定义用户界面。

1) 定义 XML 布局文件

在 Eclipse 的 Package Explorer 中，展开 res/layout/文件夹并打开 main.xml 文件，查看里面的组件定义是否符合要求，这里还是沿用原来的 main.xml。

如果在项目中有多个 Activity，需要使用不同的界面布局，可以在 res/layout/目录下创建新的布局文件。



2）在资源文件 string.xml 中定义字符串变量的值

进入 res/values/文件夹，打开 strings.xml 文件，这是用户界面保存所有默认字符串的文件。前面使用 Eclipse 工具创建 Android 项目时，ADT 已经根据创建 Android 项目时输入的信息，设定好两个字符串变量 hello 和 app_name 的值，分别为“你好，移动世界!”和“Hello Mobile World!”，见代码 A.4。

代码 A.4　string.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

<string name="hello">你好,移动世界!</string>
<string name="app_name">Hello Mobile World !</string>

</resources>
```

3）修改 HelloWorldActivity.java 文件

恢复 HelloWorldActivity.java 原来没有定义 TextView 时的代码：

```
setContentView(R.layout.main);
```

原来传递给 setContentView()方法的是一个 View 对象，现在传递的是标识为 R.layout.main 的布局资源引用。R.layout.main 实际上是一个编译好的对象，也就是在 /res/layout/main.xml 文件中的布局定义。Eclipse 插件自动在项目的 R.java 类中创建了这个引用。

重新运行 HelloWorldActivity.java 文件(见代码 A.5)，查看 AVD 上的结果。

代码 A.5　HelloWorldActivity.java

```
import android.app.Activity;
import android.os.Bundle;
public class HelloWorldActivity extends Activity {
/** Called when the activity is first created. * /
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
}
}
```

运行代码 A.5 后，显示的结果与前面使用编程式布局定义没有什么不同。下面修改一下 main.xml 文件，添加一个 TextView 定义，看看 main.xml 文件如何对应用程序布局的影响。

4）修改 main.xml 文件

先把 XML 文件中原来的 TextView 定义复制一份，放在它的下面。由于添加一个

TextView，要识别不同的 TextView 对象，需要添加一条语句：

```
android:id="@+id/textview1"
```

android:id 属性给 TextView 元素分配了一个唯一的标识 textview1。可以使用这个标识在 Java 源代码或者其他 XML 资源声明中引用这个对象。给这两个 TextView 的标识分别设置为 textview1 和 textview2，见代码 A.6。

代码 A.6　修改后的 main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical">

    <TextView
      android:id="@+id/textview1"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/hello" />
    <TextView
      android:id="@+id/textview2"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/hello" />
</LinearLayout>
```

保存 main.xml 文件。再次运行 HelloWorldActivity.java 文件，查看 AVD 界面，结果显示了两行的字符，每一行是一个 TextView。

5）修改资源文件

进入 res/values/文件夹，打开 strings.xml 文件，这是 Android 为用户界面保存所有默认字符串的地方，找到 string 的定义，这里的值就是显示在屏幕上的字符串。

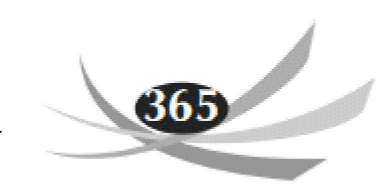
在 string 下面定义一个新的字符串 newhello，赋值为“你好，Android 移动商务平台！”。保存 strings.xml 文件。

```
<string name="newhello">你好,Android 移动商务平台！</string>
```

6）修改 main.xml 文件

修改 textview1 的 text 为 newhello，让 textview2 显示 newhello 定义的字符串，保存 main.xml 文件。代码如下：

```
<TextView
    android:id="@+id/textview2"
```




```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/newhello" />
```

再次运行 HelloWorldActivity. java 文件，AVD 上的结果显示了三行字符："Hello Mobile World""你好，移动世界！"和"你好，Android 移动商务平台！"。

7）查看 apk 文件

到这里，就完成了第一个 Android 应用程序的开发和运行，也得到了可以在真正 Android 系统上运行的 apk 文件。

展开 hello 项目中的 bin 目录，可以看到 hello. apk 文件，在手机上像安装其他 Android 程序一样安装 hello. apk 文件后，在 Android 的应用程序目录下找到 hello 的项目图标，就可以运行了。

在仿真器上调试通过后，apk 文件发布到支持相应 Android SDK 版本的手机上可以直接运行，不必再针对具体的型号进行调试，开发者只需要做到这一步就完成了 Android 应用程序的开发。

A. 3　Android 项目结构分析

A. 2 节中创建了第一个 Android 项目，下面以这个应用为基础进行目录结构概述（见图 A. 8）。

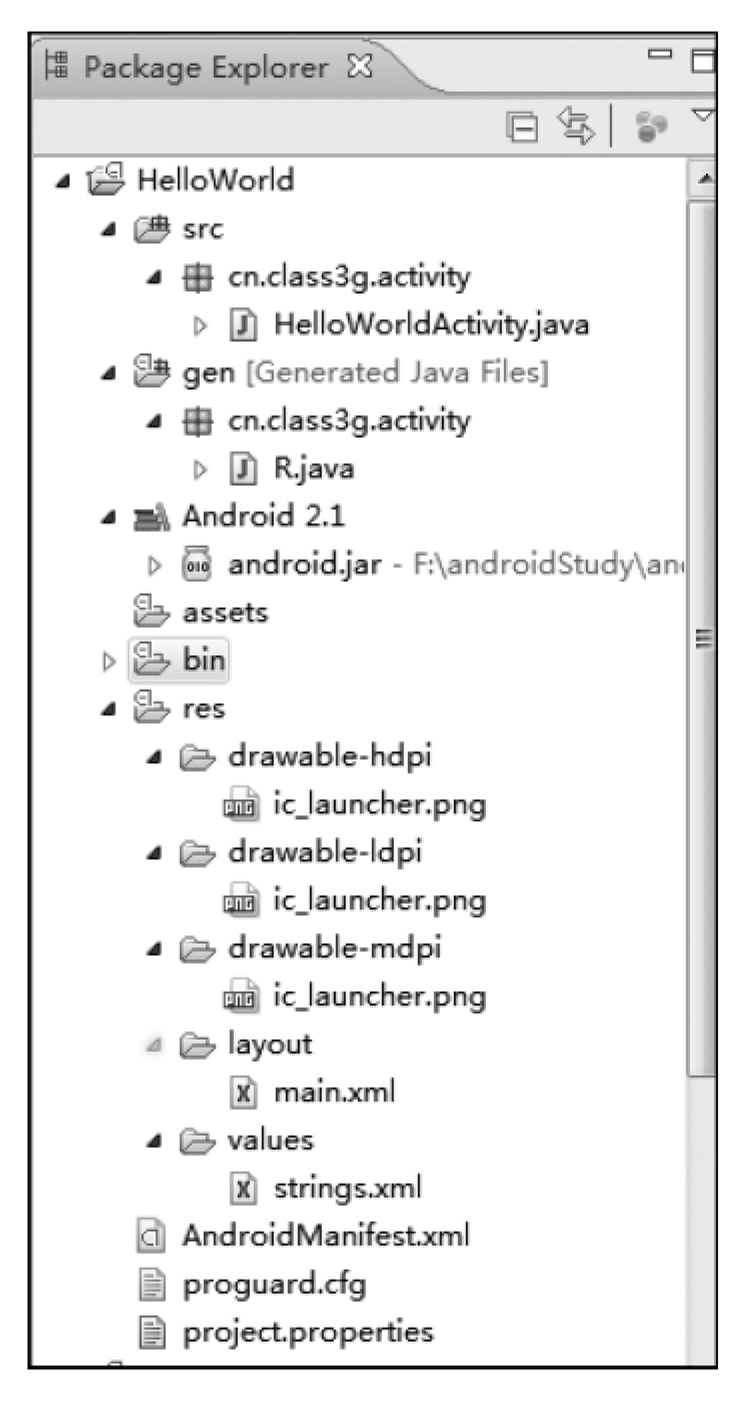


图 A. 8　Android 项目目录结构

任何 Android 项目创建后，都会自动生成图 A.8 所示的目录。下面简单介绍这些目录所包含的内容和作用。

1）src 文件夹

与一般的 Java 项目一样，src 文件夹是项目的所有包及源文件(.java)。

2）gen 文件夹

该目录下的所有文件都不是由开发人员创建的，而是由 ADT 自动生成的。其中，R.java 文件是定义该项目所有的资源文件的索引文件，该文件是只读模式。R.java 文件中默认有 attr、drawable、layout、string 四个静态内部类，其内容见代码 A.7。

代码 A.7　R.java

```
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

3）Android2.1

它包含了 android.jar 包，是 Android SDK 中提供 Android platforms API 的基本类包，Android 库、系统镜像、样本代码、仿真 skins（界面）以及指定版本的工具。这是 Android 应用程序运行的基础支持。

4）assets 文件夹

Android 系统为每个新设计的程序提供了/assets 目录，这个目录保存的文件可以打包在程序里。/res 和/assets 的不同点是，android 不为/assets 下的文件生成 ID。如果使用/assets 下的文件，需要指定文件的路径和文件名。

5）bin 文件夹

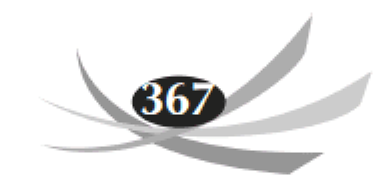
它是程序自动生成的应用文件夹，用来存放项目中应用程序生成的 apk 文件。

6）res 文件夹

该目录为资源目录，而 res 就是 resources 的缩写。该目录用来存放图标、界面文件和应用中用到的文字信息。这个目录中又有专门的子目录存放特定资源，其中：

(1) res/目录下的三个 drawable 文件夹。

这三个目录的区别只是将图标按分辨率高低来存放入不同的目录中，其中 drawable-hdpi 用来存放高分辨率图标；drawable-mdpi 用来存放中等分辨率图标；drawable-ldpi 用



来存放低分辨率图标。

(2) res/values/。

该目录用来存放strings.xml等类似于定义属性值的资源文件。

strings.xml用来定义字符串和数值，HelloWorld项目中的strings.xml文件内容如下：

```
<?xml version="1.0" encoding="utf-8" >
<resources>
    <string name="hello">Hello World World, HelloWorldActivity!</string>
    <string name="app_name">HelloWorld</string>
</resources>
```

其中，string标签声明一个字符串，name属性指定其引用名。把应用中出现的文字单独存放在strings.xml文中的原因有两个：一是为了国际化；二是为了减少应用体积，降低数据冗余。

(3) res/layout/。

该目录用来存放布局文件。main.xml布局文件是Android项目生成时自动产生的主界面布局文件，程序员可以根据自己的界面设计，定义命名自己的布局文件。

7) 根目录下的三个重要文件

AndroidManifest.xml文件：包含了该Android项目中所有使用的Activity、Service、Receiver等组件的声明。

default.properties文件：记录项目中所需要的环境信息，如Android的版本等。

proguard.cfg文件：混淆代码的脚本配置文件。

参 考 文 献

[1] Getting Started [OL]. http://developer. android. com/training/index. html.
[2] Building Apps with Content Sharing [OL]. http://developer. android. com/training/building-content-sharing. html.
[3] Best Practices for Permissions and Identifiers [OL]. http://developer. android. com/train-ing/best-permissions-ids. html.
[4] Building Apps with Location & Maps [OL]. http://developer. android. com/train-ing/building-location. html.
[5] Best Practices for User Interface [OL]. http://developer. android. com/training/best-ux. html.
[6] Best Practices for User Input [OL]. http://developer. android. com/training/best-user-input. html.
[7] About SQLite [OL]. http://www. sqlite. org/.
[8] Google Maps for every platform[OL]. https://developers. google. com/maps/.